Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Gašperlin

Razvoj informacijskega sistema s tehnologijo REST in orodjem Gradle

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Sebastijan Šprager

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Preučite tehnologijo REST in prikažite potek razvoja informacijskega sistema s smiselno funkcionalnostjo, ki temelji na tej tehnologiji. Pri gradnji informacijskega sistema uporabite orodje Gradle, katerega dobro preučite in ga tudi primerjajte z ostalimi sorodnimi rešitvami. Za implementacijo informacijskega sistema uporabite tehnologije platforme Java EE. Prav tako raziščite in realizirajte možne rešitve za ustrezno namestitev informacijskega sistema ter jih predstavite. Razvito informacijsko rešitev ustrezno ovrednotite.

Zahvaljujem se mentorju doc. dr. Sebastijanu Špragerju za nasvete in pomoč pri izdelavi diplomske naloge. Zahvaljujem se tudi profesorjem fakultete računalništva in informatike za pridobljena znanja. Zahvaljujem se še vsem prijateljem in bližnjim za podporo.

Somewhere, something incredible is
waiting to be known.
- Carl Sagan

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Tehnologija REST **5**

2.1 Protokol HTTP 6

2.2 Aplikacijski programski vmesnik 9

2.3 Primerjava s SOAP 15

3 Orodje Gradle **17**

3.1 Projekt 17

3.2 Modularna struktura 19

3.3 Naloge 20

3.4 Primerjava z drugimi orodji 21

4 Zahteve informacijskega sistema **25**

4.1 Vsebinske zahteve 25

4.2 Infrastrukturne zahteve 28

5 Uporabljene tehnologije **29**

5.1 Platforma Java EE 29

5.2 Gradle razširitve 34

5.3 Podatkovna baza PostgreSQL 36

5.4 Avtentikacijski strežnik Keycloak 36
5.5 Aplikacijski strežnik Wildfly 38
5.6 Ogrodje KumuluzEE 38
5.7 Ogrodje React 39
5.8 Obratni namestniški strežnik Nginx 40

6 Razvoj informacijskega sistema 43
6.1 Implementacija funkcionalnosti 43
6.2 Namestitev infrastrukture 63

7 Vrednotenje razvite rešitve 75
7.1 Uspešnost razvoja 75
7.2 Skaliranje 76
7.3 Nadaljnji razvoj 86

8 Sklep 89

Literatura 92

Odseki

2.1 Primer HTTP GET zahtevka 7
2.2 Primer HTTP odgovora 8
2.3 Primer HTTP GET zahtevka s parametri poizvedbe 10
2.4 Primer vsebinskega pogajanja na REST vmesniku za JSON format zahtevka in XML format odgovora 12
2.5 Primer dokumentacije REST vmesnika po specifikaciji OpenAPI 2.0 14
3.1 Primer vsebine datoteke *build.gradle* za gradnjo Java projekta s knjižnico javaee-api 18
3.2 Sprememba strukture imenikov v Gradle razširitvi Java 19
3.3 Modularna struktura Gradle projekta za Java poslovno aplikacijo 19
3.4 Vsebina korenske datoteke *build.gradle* pri modularni strukturi projekta, s katero definiramo skupne lastnosti modulov . . 20
3.5 Definicija Gradle naloge 20
3.6 Naloga *runKumuluzEE* za zagon mikrostoritve KumuluzEE . . 21
3.7 Vsebina datoteke *build.xml* za gradnjo z orodjem Ant 22
3.8 Vsebina datoteke *pom.xml* za gradnjo z orodjem Maven 23
5.1 Primer začetne točke REST vmesnik s predpono poti */api* . . 31
5.2 Primer Java razreda z JAX-RS anotacijami, ki predstavlja vir izdelka 31
5.3 Primer Java razreda z JPA anotacijami, ki predstavlja tabelo izdelkov 32

5.4 Primer Java razreda z EJB anotacijami, ki predstavlja poslovno zrno 34
6.1 Modularna struktura Gradle projekta s prevodi 44
6.2 Vsebina datoteke *settings.gradle*, ki definira relativne poti korenskih imenikov posameznih modulov 45
6.3 Abstraktni razred BaseEntity 46
6.4 Abstraktni razred BaseEntityVersion 47
6.5 Definicija poslovnega zrna baze in njegova uporaba z anotacijo @EJB 54
6.6 Razred ProductResource, ki predstavlja vir izdelkov, z implementiranima metodama *getDatabaseService* in *getAuthorizedEntity* 56
6.7 Vsebina datoteke *web.xml*, v kateri definiramo Keycloak področje (*realm*) ter varnostni vlogi *CUSTOMER* in *ADMINISTRATOR* 58
6.8 Definicija odvisnosti modula *poslovni aplikacijski arhiv EAR* . 59
6.9 Nastavitve Gradle razširitve Ear za ustvarjanje datoteke *application.xml*, ki je potrebna za namestitev arhiva EAR na aplikacijski strežnik 60
6.10 Odvisnosti modula *mikrostoritev*, s katerimi dodamo posamezno komponento ogrodja KumuluzEE 61
6.11 JavaScript implementacija funkcionalnosti za pridobitev izdelkov iz razvitega REST vmesnika 62
6.12 Nastavitev podatkovnega vira za aplikacijski strežnik Wildfly . 67
6.13 Vsebina datoteke *build.gradle* modula *microservice* z nalogama *buildKumuluzEE* in *runKumuluzEE* 69
6.14 Vsebina nastavitvene datoteke *nginx.conf* strežnika Nginx . . 72
7.1 Primer definicije identifikatorja z lastnostjo *id* tipa UUID . . . 83

Slike

3.1 Primerjava orodji Gradle, Maven in Ant 24

4.1 Vsebinske zahteve prikazane z UML diagramom 26

6.1 Zasnovana infrastruktura informacijskega sistema. 44
6.2 Podatkovna shema informacijskega sistema 48
6.3 Prikaz generične zasnove abstraktnih razredov za implementacijo virov REST vmesnika 52
6.4 Prikaz porazdeljevanja zahtevkov na strežniku Nginx 70

7.1 Prikaz testnega okolja in komunikacije med obratnim namestniškim strežnikom Nginx, aplikacijskim strežnikom Wildfly, mikrostoritvijo KumuluzEE in podatkovno bazo PostgreSQL . 79
7.2 Grafa prikazujeta predvideno število obdelanih zahtevkov na sekundo (*Number of estimated transactions/sec*) v odvisnosti s številom sočasnih uporabnikov (*Number of active threads*) za aplikacijski strežnik Wildfly in mikrostoritev KumuluzEE . 80
7.3 Graf prikazuje predvideno število obdelanih zahtevkov na sekundo (*Number of estimated transactions/sec*) v odvisnosti s številom sočasnih uporabnikov (*Number of active threads*) za obratni namestniški strežnik Nginx, ki zahtevke porazdeljuje med aplikacijski strežnik Wildfly in mikrostoritev KumuluzEE 81
7.4 Skaliranje celotne in delne podatkovne baze z dvema razdeljevalnikoma povezav 84

Seznam uporabljenih kratic

kratica	angleško	slovensko
Java EE	Java Platform, Enterprise Edition	Java platforma, poslovna izdaja
JDK	Java development kit	Java razvojno orodje
JRE	Java runtime envirioment	Java izvrševalno okolje
IS	Information system	Informacijski sistem
API	Application programming interface	Aplikacijski programski vmesnik
REST	Representational state transfer	Prenos predstavitvenega stanja
UML	Unified modeling language	Univerzalni modelni jezik
JAX-RS	Representational state transfer	Prenos predstavitvenega stanja
JPA	Java Persistence API	Javain obstojnostni vmesnik
ORM	Object-Relation Mapping	Objektno-relacijsko preslikovanje
EJB	Enterprise Java Beans	Poslovna zrna
JWT	JSON Web Token	JSON Spletni žeton
JSON	JavaScript Object Notation	JavaScript objektna notacija
BSON	Binary JavaScript Object Notation	Binarna JavaScript objektna notacija
XML	Extensible Markup Language	Razširljiv označevalni jezik
HTTP	Hyper-Text Transfer Protocol	Hipertekstni prenosni protokol
HTML	Hyper-Text Markup Language	Hipertekstni označevalni jezik
JAR	Java Archive	Java arhiv
WAR	Web Application Archive	Arhiv spletne aplikacije

EAR	Enterprise Application Archive	Arhiv poslovne aplikacije
JVM	Java virtual machine	Navidezni stroj Java
SQL	Structured Query Language	Strukturirani povpraševalni jezik
DSL	Domain specific language	Domensko specifičen jezik
DNS	Domain name service	Storitev za razreševanje domenskih imen
IP	Internet Protocol	Internetni protokol
UUID	Universally unique identifier	Univerzalno unikaten identifikator
GUID	Globally Unique Identifier	Globalno unikaten identifikator
JDBC	Java Database Connectivity	Bazna povezljivost Java
JPQL	Java Persistence Query Languagey	Java obstojnostni povpraševalni jezik
CDI	Context and Dependency Injection	Vstavljanje konteksta in odvisnosti
AWS	Amazon web services	Amazon spletne storitve
SOA	Service Oriented Architecture	Storitveno usmerjena arhitektura

Povzetek

Naslov: Razvoj informacijskega sistema s tehnologijo REST in orodjem Gradle

V diplomski nalogi predstavimo razvoj informacijskega sistema s tehnologijo REST. Potek razvoja prikažemo na praktičnem primeru spletne trgovine. Funkcionalnost je implementirana v programskem jeziku Java in tehnologijami platforme Java EE. Z orodjem Gradle je zasnovana fleksibilna in modularna struktura projekta, ki nam omogoča enostavno nadgradnjo, menjavo in ponovno uporabo posameznih komponent. Predstavimo in integriramo tudi tehnologije potrebne za postavitev infrastrukture poslovne aplikacije. Hkrati prikažemo enostaven prestop na arhitekturo mikrostoritev, pri čemer ponovno uporabimo module poslovne aplikacije. Razviti informacijski sistem nazadnje ovrednotimo glede na fleksibilnost uporabljenih tehnologij in preverimo njihovo učinkovitost pri skaliranju.

Ključne besede: informacijski sistem, REST, razvoj, Gradle, Java EE.

Abstract

Title: Development of information system with REST technology and Gradle build tool

In this thesis we present a development procedure of information system using REST technology. Approach is demonstrated on a practical example of online shop. Functionality is implemented in programming language Java and employs technologies of Java EE platform. With Gradle build tool we design flexible and modular project structure, which enables to easily upgrade, change or reuse individual components. We introduce and integrate technologies needed for setting up business application infrastructure. We also show a simple transition to microservice arhitecture in which we reuse modules of business application. Finally, we evaluate the developed information system based on the flexibility of used technologies and examine the efficiency of scalability.

Keywords: information system, REST, development, Gradle, Java EE.

Poglavje 1

Uvod

Postopki razvoja programske opreme se neprestano spreminjajo in prilagajajo potrebam trga. Podjetja stremijo po odzivnih, robustnih, skalabilnih in funkcijsko obsežnih informacijskih rešitvah, ki jim dajejo konkurenčno prednost. Informacijske rešitve zato v zadnjih letih postajajo vse bolj obsežne in zapletene, kar predstavlja velik izziv za razvojne ekipe. Izvorna koda v takih projektih lahko hitro postane težko obvladljiva, kar ogrozi uspešnost projekta. Za lažje vodenje projektov so se pojavile nove metodologije razvoja, kot sta Scrum in Kanban [29]. Razvili so se tudi novi programski jeziki, orodja in tehnologije, ki nam omogočajo večjo fleksibilnost pri razvoju informacijskih rešitev.

Razvojne ekipe morajo dobro poznati zahteve naročnika in izbrati tehnologije, ki najbolje rešujejo zadane probleme. V praksi veliko število projektov ni v celoti uspešnih oz. se čas razvoja drastično poveča, zaradi ovir pri razvoju [19]. Dobro poznavanje prednosti in slabosti izbranih tehnologij je ključno za uspešnost projekta. Kljub temu se s problemi srečujejo tudi izkušene razvojne ekipe, saj vseh problemov pri razvoju večjih projektov praktično ni mogoče predvideti.

Tehnologija REST, katere glavne prednosti sta skalabilnost in enostaven razvoj, je ena izmed najučinkovitejših, kar je razlog za njeno hitro adaptacijo. Njena uporaba je danes prisotna v vseh večjih podjetjih, kot so Amazon, Fa-

cebook, Google, itd. Ta kot prenosni protokol uporablja HTTP, s katerim enostavno implementiramo bogat aplikacijski vmesnik. Večina programskih ekosistemov že podpira orodja za razvoj REST storitev, kot so ASP.NET Web API (C#), Django REST (Python), Lumen (PHP) itd. Najpopularnejši in najbolj razširjen programski jezik za razvoj informacijskih rešitev v poslovnem svetu je Java s tehnologijami platforme Java EE [32], zato bo tudi razvoj informacijskega sistema potekal v tem programskem jeziku. Glavni razlogi za njegovo izbiro so predvsem velik nabor odprtokodnih knjižnic, dobra podpora pri integriranih razvojnih okoljih ter obsežna uporabniška skupnost. Java je pogosto glavni programski jezik v večjih podjetjih, saj je tehnologija dobro preizkušena ter ima veliko izobraženega kadra. Projekti informacijskih sistemov lahko postanejo zelo obsežni, zato je dobra struktura projekta ključna za obvladovanje izvorne kode. Ta problem rešimo z uporabo orodja Gradle, ki z domensko specifičnim jezikom (*domain specific language*) nudi visok nivo fleksibilnosti ter omogoča uporabo razširitev, s katerimi zlahka dodamo potrebno funkcionalnost za vzdrževanje projekta.

V diplomski nalogi pričnemo z opisom tehnologije REST, na kateri temelji informacijski sistem. Nadaljujemo s podrobnejšo predstavitvijo ogrodja Gradle in njegove razširitve Java, s katero gradimo in testiramo Java projekte. Temu sledi opis zahtev informacijskega sistema, ki smo jih razdelili na vsebinske in infrastrukturne. Vsebinske zahteve opisujejo potrebno funkcionalnost, ki jo prikažemo z UML diagramom, za stranke in administratorje, medtem ko infrastrukturne zahteve opisujejo zmogljivosti sistema. Za implementacijo informacijskega sistema potrebujemo obsežen nabor tehnologij, ki so med seboj združljive in skupaj v celoti izpolnjujejo zadane zahteve. Te podrobneje preučimo, nakar pričnemo z razvojem informacijske rešitve. Sprva smo zasnovali modularno strukturo projekta z orodjem Gradle, ki se deli na osnovne in aplikacijske module. Osnovni moduli se delijo na *shema baza*, *poslovna logika* in *REST komponente*, ki jih ponovno uporabimo pri aplikacijskih modulih *strežnik*, *mikrostoritev* in *vmesnik*. S tem smo dosegli fleksibilno strukturo in poenostavili obvladovanje izvorne kode. Posameznim modulom smo z

orodjem Gradle in njegovimi razširitvami dodali še potrebno funkcionalnost za gradnjo, testiranje, namestitev in zagon. V primeru modula *shema baze* smo dodali še podporo migracij podatkovne baze. Po končani implementaciji funkcionalnosti smo pričeli z namestitvijo in integracijo tehnologij potrebnih za postavitev izvajalne infrastrukture poslovne aplikacije, med katere sodijo podatkovna baza, avtentikacijski strežnik, aplikacijski strežnik, mikrostoritev ter obratni namestniški strežnik (*reverse proxy server*). S podatkovno bazo vzpostavimo podatkovni vir, ki ga uporabljata aplikacijski strežnik in mikrostoritev. Prijave uporabnikov in njihovo upravljanje bo potekalo preko avtentikacijskega strežnika s protokolom OAuth 2. Da uporabnikom omogočimo dostop do vseh potrebnih storitev preko enotnega naslova, uporabimo obratni namestniški strežnik, ki bo hkrati deloval kot predpomnilnik in razporejevalnik zahtevkov. Po končani implementaciji smo razvito informacijsko rešitev ustrezno testirali in ovrednotili. Predstavili smo tudi uspešnost implementacije zadanih zahtev informacijskega sistema ter njegovo fleksibilnost. Pri tem smo ovrednotili tudi skalabilnost sistema ter testirali še posamezno in skupno zmogljivost aplikacijskega strežnika in mikrostoritve. Na koncu smo identificirali in predstavili možne izboljšave pri nadaljnjem razvoju informacijskega sistema.

Poglavje 2

Tehnologija REST

Arhitekturni stil Representational State Transfer (REST) je bil prvič predstavljen leta 2000 v doktorskem delu Architectural Styles and the Design of Network-based Software Architectures [16], katerega avtor je Roy Thomas Fielding. Fielding opiše arhitekturo spletnih storitev za porazdeljene hipermedijske sisteme. Tehnologija je bila razvita z namenom visoke zmogljivosti, skalabilnosti, preprostosti, prenosljivosti in zanesljivosti. Te lastnosti so tudi razlog za naslednje omejitve:

- Komunikacija med strežnikom in odjemalcem mora potekati brez stanj (*stateless*), kar razbremeni strežnik in poveča njegovo zmogljivost. Temu se morajo prilagoditi tudi zahtevki, saj morajo vsebovati vse podatke za izvedbo želene operacije. Izboljša se tudi preglednost monitoriranja sistema, saj nam ni potrebno spremljati zgodovine, ampak le posamezni zahtevek. Slabost takega pristopa je, da morajo stanje hraniti in usklajevati aplikacije na strani odjemalca, nad čemer strežnik nima nadzora.

- Ločitev zahtev uporabniškega vmesnika od zahtev shrambe podatkov, kar omogoča neodvisen razvoj obeh komponent.

- Uporaba protokola HTTP omogoča predpomnjenje odgovorov na zahtevo z metodo GET, kar zmanjša čas odzivnosti in obremenjenosti sis-

tema. Pri tem moramo ustrezno nastaviti osveževanje predpomnilnika, da podatki ne zastarajo.

- Univerzalen vmesnik med komponentami poenostavi arhitekturo sistema in interakcijo posameznih komponent. Pri tem je potrebno upoštevati, da se poslabša učinkovitost, saj se sporočila prenašajo v standardizirani obliki, ki ni primerna za vse vrste podatkov.

- Tehnologija REST omejuje komunikacijo med različnimi sloji sistema. Posamezen sloj lahko komunicira le med sosednjimi sloji, kar zmanjša kompleksnost celotnega sistema. V nadaljevanju razvoja takega sistema ni težko spreminjati in razširjati.

- Strežnik lahko odjemalcu ponudi razširitev v obliki izvršljive kode (Code-On-Demand). Primer take kode je JavaScrip ali Java applet, ki se brez namestitve lahko izvaja na odjemalčevem računalniku. Prenesena koda vsebuje ključne funkcionalnosti, katerih ni ponovno potrebno implementirati.

2.1 Protokol HTTP

Tehnologija REST za prenosni protokol uporablja HTTP, ki se je skozi leta spreminjal in nadgrajeval za potrebe spleta. Prva verzija HTTP/1.0 je bila objavlena leta 1996 (RFC 1945) [5]. Sprva je vseboval le metode GET, POST in HEAD. V naslednji verziji HTTP/1.1 (RFC 2068) [13] izdana leto kasneje so bile dodane še metode OPTIONS, PUT, DELTE in TRACE, ki so ključni gradniki REST vmesnika. Dodani sta bili še metodi PATCH (RFC 5789) [10] in CONNECT (RFC 2616) [15]. Zadnja različica protokola HTTP/2 je bila definirana leta 2015 (RFC 7540) [4]. Ta omogoča boljšo zmogljivost in pri tem ohranja obstoječo obliko sporočil.

Danes protokol HTTP vsebuje 8 različnih metod, če ne upoštevamo metode PATCH, ki uradno ni del specifikacije (RFC 7231) [14]. Z njimi določimo tip operacije, ki jo želimo izvesti z zahtevkom. Spletnemu viru lahko vsako

metodo implementiramo le enkrat. Da spletni vir ostane pregleden, se držimo že ustaljenih praks za vsako izmed naslednjih metod:

- GET - pridobi vsebino na želenem viru,
- HEAD - pridobi vrednosti GET zahtevka brez telesa,
- POST - pošlje telo s podatki, ki so obdelani na viru,
- PUT - pošlje telo s podatki, ki spreminjajo že obstoječe podatke na strežniku,
- PATCH - spremeni podatke vsebovane v telesu zahtevka,
- DELETE - briše želene podatke na viru,
- OPTIONS - pridobi metode, ki jih vir podpira,

Za dostop do REST vmesnika je potreben HTTP zahtevek, ki je sestavljen iz naslednjih delov:

- Zahtevna vrstica (*request line*), ki sestoji iz:
  - Metoda (*Method*) zahtevka,
  - Ime vira (*Uniform Resource Identifier*),
  - Verzija (*Version*) HTTP protokola,
- Glava (*Head*) vsebuje različne podatke o zahtevku, ki so v obliki ključa in vrednosti (*key/value*),
- Telo (*Body*) je možno le pri metodah POST, PUT, CONNECT in PATCH. Vsebuje podatke potrebne za obdelavo zahtevka,

Primer GET zahtevka je prikazan v odseku 2.1.

```
GET /index.html HTTP/1.1
User-Agent: Mozilla/4.0
Host: www.fri.com
Accept-Language: en-us
Connection: Keep-Alive
```

Odsek 2.1: Primer HTTP GET zahtevka

HTTP strežnik na prejeti zahtevek vrne odgovor v vsakem primeru, tudi kadar pride do napake. Za lažje razumevanje napake oz. odgovora ima HTTP protokol definirana stanja. Z njimi natančneje opišemo uspešnost obdelanega zahtevka ali napako. Vsako stanje vsebuje frazo in kodo, ki je sestavljena iz treh števk. Prva števka predstavlja enega izmed naslednjih tipov skupin:

- 1xx - informacije (*informational*)
- 2xx - uspešno (*success*)
- 3xx - preusmeritev (*redirection*)
- 4xx - napaka na odjemalcu (*client error*)
- 5xx - napaka na strežniku (*server error*)

HTTP odgovor je po strukturi podoben zahtevku, razlikujeta se le v prvi vrstici. Zahtevek vsebuje podatke o viru in metodi, medtem ko odgovor nosi podatke o stanju. Struktura odgovora je tako sledeča:

- Statusna vrstica (*status line*), ki je sestavljena iz:
  - Verzija (*version*) HTTP protokola,
  - Koda stanja (*status code*),
  - Fraza stanja (*reason phrase*),
- Glava (*head*) vsebuje podatke o odgovoru v obliki ključa in vrednost.
- Telo (*body*) nosi podatke odgovora, najpogosteje v obliki HTML ali JSON.

Primer HTTP odgovora s statusom OK je prikazan v odseku 2.2.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Connection: close

<html>}
<body>
  Hello World.
</body>
</html>
```

Odsek 2.2: Primer HTTP odgovora

2.2 Aplikacijski programski vmesnik

Aplikacijski programski vmesnik (*application programming interface*) oz. API se uporablja za izpostavljanje ključnih operacij, s katerimi upravljamo kompleksnejšo tehnologijo. Namenjen je razvijalcem, katerim poenostavi integracijo nove tehnologije v njihovo informacijsko rešitev. Prisotnost API-jev lahko opazimo na vseh področjih razvoja informacijskih rešitev, saj bi se brez njih precej povečala kompleksnost integracije obstoječih rešitev. Poznamo mnogo tipov API-je, med katerimi so najbolj razširjene programske knjižnice. Mi se osredotočimo na spletne API-je podrobneje REST API-je. Ti izpostavljajo funkcionalnost sistema z viri, ki jih podrobneje preučimo v naslednjem podpoglavju.

2.2.1 Viri

Implementacijo REST aplikacijskega programskega vmesnika je potrebno skrbno načrtovati, saj mora biti čim bolj razumljiv vsakemu razvijalcu. Pot vira izpeljemo iz domene oz. tabele, katero vir izpostavlja. Dobra praksa zasnove vira je po načelu CRUD [33] (*create*, *read*, *update*, *delete*), ki predstavlja osnovne štiri operacije za manipulacijo podatkovne baze. Vsaki operaciji pripišemo pomensko enako metodo. Za primer izdelka v spletni trgovini bi dobili sledečo strukturo metod:

- GET */Product*
- GET */Product/*{*id*}
- POST */Product*
- PUT */Product/*{*id*}
- PATCH */Product/*{*id*}
- DELETE */Product/*{*id*}

Operacijo branja predstavlja metoda GET, ki implementira dve različni poti. Pot brez argumenta *id* vrne seznam izdelkov in omogoča iskanje, sortiranje ter paginacijo. Izdelke lahko tudi iščemo po vrednosti njenih lastnosti ter jih poljubno uredimo. Za pridobitev prve strani izdelkov velikosti 20, ki imajo vrednost cene manjše od 5 in so urejeni padajoče po ceni, uporabimo zahtevek prikazan v odseku 2.3.

```
GET /Product?limit=20&skip=0&
where=price:lt:5&order=price DESC
```

Odsek 2.3: Primer HTTP GET zahtevka s parametri poizvedbe

Kadar želimo pridobiti le en izdelek uporabimo drugo pot metode GET, ki ji podamo argument *id* oz. identifikator. Ta enolično določa entiteto v bazi, ki je najpogosteje celo število (*int*), ki lahko predstavlja 2^{32} različnih entitet, ali UUID, ki se ustvari naključno in lahko zavzema 2^{122} različnih vrednosti. Nov izdelek ustvarimo z zahtevkom POST, katerega telo vsebuje vse potrebne podatke izdelka v podprtem formatu. Strežnik zahtevku POST odgovori z identifikatorjem v glavi *Location*, ki ga je baza določila ustvarjenemu izdelku. Obstoječi izdelek lahko nato spremenimo z zahtevkom PUT ali PATCH. Oba zahtevka potrebujeta identifikator izdelka, ki ga spreminjata, in telo z novimi podatki. Ključna razlika zahtevkov je v načinu spreminjanja podatkov. Ob uporabi zahtevka PUT bodo lastnosti izdelka, ki jih v telesu nismo navedli, privzele prazno vrednost, medtem ko zahtevek PATCH posodobi samo lastnosti navedene v telesu zahtevka. Kadar želimo posodobiti le eno lastnost z uporabo metode PUT, moramo v telo vključiti tudi vse nespremenjene lastnosti. Telo tako nosi odvečne podatke, kar poveča obremenitev omrežja, zato se v takih primerih priporoča uporaba metode PATCH. Izdelek izbrišemo z zahtevkom DELETE, ki tako kot PUT in PATCH zahteva identifikator izdelka, ki ga želimo izbrisati.

2.2.2 Verzioniranje

Verzioniranje vmesnika ni potrebno, ampak če nameravamo vmesnik tekom življenjske dobe tudi spreminjati je verzioniranje obvezno, saj le tako lahko uporabnikom zagotovimo postopni prehod. Trenutno v skupnosti ni standardnega načina verzioniranja, zato o tem odločajo razvijalci na podlagi svojih izkušenj. Najbolj so razširjeni naslednji načini:

- URL - verzijo podamo v URL-ju najpogosteje takoj za oznako vmesnika (*/api/v1/Product*),
- parameter poizvedbe - navedemo ga na koncu spletnega naslova z vrednostjo verzije (*/api/Product?version=1*),
- lastnost v glavi - verzijo sporočimo v glavi HTTP zahtevka z novo lastnostjo (*"X-API-Version": "1"*),
- zahtevan tip vsebine - definiramo nov tip vsebine (*"Accept": "application/vnd.domain.v1+json"*),

Vsak način ima svoje prednosti in slabosti. Večja podjetja, kot sta Google in Amazon, uporabljajo URL pristop, medtem ko se je Facebook odločil za parameter poizvedbe. Podjetje Stripe poleg verzije v URL-ju uporablja tudi glavo *Stripe-Version* z vrednostjo datuma izdaje nove verzije, ki je združljiva s prejšnjo verzijo. Kadar pride do večjih sprememb, ki niso več združljive s predhodno verzijo, se spremeni tudi verzija v URL-ju.

2.2.3 Vsebinsko pogajanje

Tehnologija REST ponuja robustnost tudi v formatu sporočila. Z uporabo glav *Content-Type* in *Accept* lahko izbiramo med različnimi formati vsebine telesa. Glava *Content-Type* določa format podatkov v telesu zahtevka, medtem ko glava *Accept* določa pričakovani format podatkov v telesu odgovora. V odseku 2.4 je prikazan primer zahtevka poslanega v formatu JSON in odgovora v formatu XML.

```
POST /Product HTTP/1.1
Content-Type: application/json
Accept: application/xml

{
        "title" : "Solid state drive",
        "price" : "120"
}


HTTP/1.1 201 Created
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<product>
        <id>ac4c476b-a594-4d79-9734-e7bbfecd5ba5</id>
</product>
```

Odsek 2.4: Primer vsebinskega pogajanja na REST vmesniku za JSON format zahtevka in XML format odgovora

Najpogosteje se uporablja format JSON, saj je bolj ekonomičen in hitreje obdelan kot XML [34]. Specifikacija nam dopušča tudi implementacijo novega formata, ki ga lahko prilagodimo potrebam projekta. Definicija novega formata je v praksi redka, saj ta samo še dodatno zaplete implementacijo.

2.2.4 OpenAPI dokumentacija

Za enostavnejše odkrivanje zmogljivosti API-ja oz. REST vmesnika je tega potrebno natančno dokumentirati. Dokumentacija se najpogosteje pripravi po standardu OpenAPI [9] imenovan tudi Swagger. Strežnik tako poleg virov vsebuje še opis teh v formatu JSON, ki je v našem primeru dostopen na naslovu */api/v1/swagger.json*. Opisa takega vira izdelkov je prikazan v odseku 2.5, ki v korenu vsebuje naslednje lastnosti:

- swagger - trenutna različica specifikacije,
- info - opis vmesnika,
- host - naslov strežnika,
- basepath - osnovna pot do vmesnika,
- tags - oznake,

- schemes - protokoli za komunikacijo,
- consumes - podprti formati sporočil v zahtevku,
- produces - podprti formati sporočil v odgovoru,
- paths - opisi vseh poti vključno z metodami, parametri in oblike sporočil,
- securityDefinitions - opis prijave in varnostnih vlog,
- definitions - opisi modelov vmesnika,

```
{
  "swagger": "2.0",
  "info": { "version": "1.0", "title": "Api" },
  "host": "localhost",
  "basePath": "/api/v1",
  "tags": [ { "name": "Product" } ],
  "schemes": ["http"],
  "consumes": ["application/json"],
  "produces": ["application/json"],
  "paths": {
        /Product: {
         "get": {
           "tags": ["product"],
           "operationId": "getList",
           "consumes": ["application/json"],
           "produces": ["application/json"],
           "parameters": [],
           "responses": {
                  "200": {
                "description": "successful operation",
                "schema": {
                  "$ref": "#/definitions/Product"
                }
                 }
                }
          },
         "post": {...}
        },
        /Product/{id} : {
                "get": {...},
                "put": {...},
                "delete": {...}
        }
  },
  "securityDefinitions": {
        "o2auth": {
            "description": "Keycloak authentication using OAUTH 2.0 .",
            "type": "oauth2",
            "authorizationUrl": "http://domain/auth/realms/is",
            "flow": "implicit",
            "scopes": {
                "CUSTOMER": "Customer role.",
                "ADMINISTRATOR": "Administrator role."
            }
        }
  },
  "definitions": {
    "Product": {
      "type": "object",
      "properties": {
        "id": { "type": "integer", "format": "int32" },
        "title": { "type": "string"},
        "price": { "type": "number", "format": "double" }
      }
    }
  }
}
```

Odsek 2.5: Primer dokumentacije REST vmesnika po specifikaciji OpenAPI 2.0

2.3 Primerjava s SOAP

Storitveno usmerjena arhitektura oz. SOA [22] je arhitekturni stil poslovne aplikacije, ki izpostavlja storitve drugim aplikacijam preko spleta. Storitve tako lahko skupaj predstavljajo funkcionalnost večjih informacijskih sistemov. Arhitektura je neodvisna od tehnologije, naprav in komunikacijskega protokola, zato njeno implementacijo lahko izvedemo z različnim naborom tehnologij. Najpogosteje se ta implementira s tehnologijo SOAP ali REST. Obe tehnologiji sta dobro preizkušeni in nudita bogat nabor razvojnih orodji.

Tehnologiji sta si arhitekturno precej podobni, vendar imata nekaj večjih razlik, na podlagi katerih se odločamo za njuno izbiro pri implementaciji. Tehnologija REST izpostavlja vire, ki jih manipuliramo z operacijami CRUD, medtem ko tehnologija SOAP izpostavlja poslovno logiko. Tehnologija SOAP z razširitvijo WS-Security ponuja tudi večji nabor varnostnih funkcionalnosti, vendar se te razen v finančnemu sektorju uporabljajo le redko. Razširitev WS-AtomicTransaction tehnologiji SOAP omogoči uporabo ACID transakcij, ampak uporaba takšnih transakcije zahteva veliko virov in jih v praksi uporabljamo le kadar je to nujno potrebno (finančne transakcije). Zahtevek tehnologije SOAP je ovit v ovojnico in podpira le format XML, kar še dodatno poveča porabo virov v primerjavi s tehnologijo REST, ki nima predpisanega formata.

Danes je na spletu med javnimi spletnimi API-ji najbolj razširjena tehnologija REST, saj enostavna uporaba in implementacija omogočata hiter razvoj ter hkrati ne zahteva veliko procesnih virov. Drugače je v svetu financ, kjer je dodatna varnost in zanesljivost ključnega pomena in je uporaba tehnologije SOAP bolj ustrezna.

Poglavje 3

Orodje Gradle

Gradbeno orodje Gradle 1.0 [17] je bilo izdano leta 2012 in velja za naslednika orodij Maven [3] in Ant [2]. Orodje je napisano v programskem jeziku Java, zato za izvajanje potrebuje JRE. Primarno se uporablja za gradnjo Java projektov, vendar trenutna verzija podpira tudi preostale programske jezike, kot so C, C++ in Python. Gradle za opis gradnje projekta uporablja svoj domensko specifičen jezik oz. DSL, ki temelji na programskem jeziku Groovy. Z njim definiramo naloge in njene odvisnosti, ki predstavljajo najmanjšo izvršitveno enoto. Te definiramo v datoteki *build.gradle*, ki jo orodje Gradle pred vsako izvršitvijo naloge preveri za morebitne spremembe in po potrebi ponovno ustvari graf vseh nalog in njenih odvisnosti. Zgrajeni graf mora biti acikličen, saj v primeru ciklične odvisnosti med nalogami gradnja grafa ne uspe. Gradle projektu potrebno funkcionalnost lahko implementiramo v programskem jeziku Groovy ali dodamo z razširitvami. V nadaljevanju tega poglavja smo podrobneje predstavili vsebino projekta, primer modularne strukture ter implementacijo naloge.

3.1 Projekt

Gradle projekt je predstavljen z imenikom, ki v korenu vsebuje datoteko *build.gradle*. V njej definiramo funkcionalnost za gradnjo, testiranje, zagon in

namestitev projekta. Struktura imenikov v projektu je odvisna od uporabljenega programskega jezika in njegove Gradle razširitve. Praktično uporabo orodja Gradle podrobneje predstavimo na primeru programskega jezika Java, ki je najbolj popularen. V tem primeru dodamo stavek *apply plugin: 'java'*, ki se mora nahajati na začetku datoteke. Razširitev definira nove naloge, kot so *assemble*, *test*, *build* in *jar*. Pred njihovo uporabo je potrebno nastaviti vrednosti podedovanih lastnosti *group*, *version* ter *sourceCompatibility*. Z vrednostjo *group* nastavimo ime skupine, kateri pripada projekt. Lastnost *version* hrani vrednost verzije, ki jo pred izdajo nove verzije arhivov popravimo. Programski jezik Java se neprestano razvija, zato je potrebno v lastnosti *sourceCompatibility* specificirati verzijo prevajalnika, ki je potreben za gradnjo zložne kode. Če projekt potrebuje dodatne odvisnosti, te specificiramo v lastnosti *dependencies*, za katere je potrebno dodati ustrezna oddaljena odlagališča (*repository*). V odseku 3.1 je prikazan primer projekta, ki potrebuje knjižnico javaee-api ter uporablja odlagališče Maven Central.

```
apply plugin: 'java'

group 'si.fri.demo.is'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}
dependencies{
    dependency group: 'javax', name: 'javaee-api', version: '7.0'
}
```

Odsek 3.1: Primer vsebine datoteke *build.gradle* za gradnjo Java projekta s knjižnico javaee-api

Java projekt privzeto v korenu vsebuje imenika *build* in *src*. V imeniku *build* se hrani vsa zložna koda pripravljena za izvajanje, medtem ko se v imeniku *src* hrani vsa izvorna koda. Slednjo ločimo na glavno in testno kodo z imenikoma *main* in *test*, ki vsebujeta še imenika *resources* in *java*. Slednji predstavlja korenski imenik izvorne kode. V njem hranimo vse Java razrede, ki so razdeljeni v imenike, ki predstavljajo njihov paket. V imeniku *resources* hranimo potrebne nastavitvene datoteke, ki se skupaj z zložno kodo zapakirajo v arhiv. Predstavljena struktura imenikov ni obvezna, saj jo lahko tudi

spreminjamo, kar prikazuje odsek 3.2, ki privzeti imenik *./src/main/java* nadomesti z imenikom *./izvorna-koda*. S tem enostavno prilagajamo strukturo imenikov potrebam projekta in uporabljenih tehnologij.

```
sourceSets {
    main {
        java {
            srcDirs = ['izvorna-koda']
        }
    }
}
```

Odsek 3.2: Sprememba strukture imenikov v Gradle razširitvi Java

3.2 Modularna struktura

V prejšnjem primeru smo predstavili uporabo orodja Gradle na monolitnemu Java projektu. Monolitne arhitekture niso primerne za gradnjo obsežnih aplikacij, saj datoteka *build.gradle* hitro postane zelo obsežna, kadar potrebujemo veliko število knjižnic in nalog. Tak projekt je težko vzdrževati in praktično nemogoče zagotoviti njegovo ponovno uporabo. Ta problem rešimo z delitvijo projekta na module, ki predstavljajo funkcionalnost manjše zaključene enote in imajo svojo *build.gradle* datoteko.

```
projekt
|
+-- jpa
|   +-- build
|   +-- src
|   +-- build.gradle
+-- ejb
|   +-- build
|   +-- src
|   +-- build.gradle
+-- rest
|   +-- build
|   +-- src
|   +-- build.gradle
+-- build.gradle
+-- settings.gradle
```

Odsek 3.3: Modularna struktura Gradle projekta za Java poslovno aplikacijo

Struktura Java projektov, ki uporablja tehnologije platforme Java EE, se pogosto deli na module, ki implementirajo funkcionalnost posamezne tehnologije. V primeru razvoja REST vmesnika bi strukturo projekta lahko

razdelili na module *jpa*, *ejb* in *rest*. Tako strukturo prikazuje odsek 3.3, ki v korenskem imeniku vsebuje 3 module in 2 datoteki. V korenu zasledimo novo datoteko *settings.gradle*, v kateri definiramo relativne poti korenskih imenikov modulov. Kljub temu, da korenski imenik *projekt* ne vsebuje izvorne kode, ta potrebuje datoteko *build.gradle*, v kateri lahko definiramo skupne lastnosti vsem projektom. To dosežemo z uporabo metode allproject oz. subprojects prikazano v odseku 3.4.

```
allprojects {
    apply plugin: 'java'

    group 'si.fri.demo.is'
    version '1.0-SNAPSHOT'
    sourceCompatibility = 1.8

    repositories {
        mavenCentral()
    }
    dependencies{
        dependency group: 'javax', name: 'javaee-api', version: '7.0'
    }
}
```

Odsek 3.4: Vsebina korenske datoteke *build.gradle* pri modularni strukturi projekta, s katero definiramo skupne lastnosti modulov

3.3 Naloge

Naloge predstavljajo najmanjšo izvršitveno enoto orodja. Te k projektu lahko dodamo z uporabo razširitev ali jih implementiramo s programskim jezikom Groovy. Definicijo nove naloge prikazuje odsek 3.5.

```
task helloWorld {
    dependsOn assemble
    doFirst {
        println project.ext.sporocilo
        project.ext.sporocilo = 'Svet'
    }
    doLast {
        println project.ext.sporocilo
    }
    project.ext.sporocilo = 'Pozdravljen'
}
```

Odsek 3.5: Definicija Gradle naloge

Nalogi postavimo odvisnost na nalogo *assemble* s stavkom *dependsOn assemble*. V nadaljevanju se bo, ko bomo hoteli izvesti obravnavano nalogo, vedno izvedla naloga *assemble*, ki pripada razširitvi Java. V nalogi iz primera smo nastavili vrednost projektne spremenljivke *sporocilo* na Pozdravljeni, ki smo jo nato izpisali v metodah *doFirst* in *doLast* s stavkom *println project.ext.sporocilo*. V metodi *doFirst* smo po izpisu tudi spremenili njeno vrednost na Svet. Izvršitev omenjene naloge tako povzroči gradnjo zložne kode in izpis besed Pozdravljeni in Svet v ukazni lupini. To morda na prvi pogled ni pričakovano, zaradi vrstnega reda nastavljanja vrednosti projektne spremenljivke *sporocilo*. Privzeto naloga vsebuje prazni metodi *doFirst* in *doLast*, ki se izvedeta med izvrševanjem. V nalogi *helloWorld* smo metodi ponovno definirali pred nastavitvijo projektne spremenljivke *sporocilo*, vendar se njuna vsebina izvede po vzpostaviti. Tudi obstoječim nalogam, kot je *build*, lahko dodajamo funkcionalnosti in odvisnosti, ki niso omejene na naloge znotraj istega modula. Nalogo lahko ustvarimo tudi z uporabo tipov nalog, ki vsebujejo že definirano funkcionalnost. Tip naloge *JavaExec* pripada razširitvi Java in nam omogoča zaganjanje Java aplikacij. Njegovo implementacijo prikazuje odsek 3.6, v katerem je potrebno nastaviti lastnosti *main* in *classpath*. Z nalogami tako lahko na osnovi razširitev implementiramo bogato funkcionalnost.

```
task runKumuluzEE(type: JavaExec) {
    dependsOn buildKumuluzEE
    doFirst {}
    doLast {}
    main = 'com.kumuluz.ee.EeApplication'
    classpath = ...
    environment('PORT', '7080')
}
```

Odsek 3.6: Naloga *runKumuluzEE* za zagon mikrostoritve KumuluzEE

3.4 Primerjava z drugimi orodji

Danes je na trgu mnogo orodji, ki nam olajšujejo gradnjo projektov. To velja za vse večje programske jezike, vendar se bomo v nadaljevanju osredotočili

le na orodja, ki se uporabljajo za gradnjo Java projektov. Najbolj razširjeni med njimi so Ant, Maven in Gradle. Orodje Gradle bomo zato primerjali z orodjema Ant in Maven.

3.4.1 Primerjava z orodjem Ant

Gradbeno orodje Ant, izdano leta 2000, je bilo zasnovano v programskem jeziku Java, tako kot Gradle. Predstavljal je napredek nad orodjem Make in hitro postal primarno orodje za gradnjo Java projektov. Temu so kasneje dodali tudi podporo za razširitve, medtem ko so za upravljanje s knjižnicami uporabljali orodje Ivy. Postopek gradnje projekta je bil opisan v formatu XML in se je hranil v datoteki *build.xml*. V njej definiramo tarče (*targets*), ki predstavljajo skupek operacij in so pomensko podobne nalogam. Če bi želeli dodati funkcionalnosti Gradle razširitve Java, bi morali za vsako nalogo definirati novo tarčo s potrebnimi operacijami in med njimi vzpostaviti odvisnost. Vsako tarčo je potrebno podrobno opisati, zato vzdrževanje hitro postane kompleksno.

```
<target name="clean">
        <delete dir="build"/>
</target>

<target name="assemble">
        <mkdir dir="build/classes"/>
        <javac srcdir="src/main/java"
                    destdir="build/classes"
                    classpath="dependencies"/>
</target>

<target name="jar" depends="assemble">
        <mkdir dir="build/lib"/>
        <jar destfile="build/lib/application.jar" basedir="build/classes"/>
</target>
```

Odsek 3.7: Vsebina datoteke *build.xml* za gradnjo z orodjem Ant

Primer *build.xml* datoteke prikazuje odsek 3.7, v katerem definiramo tarče *clean*, *assemble* in *jar*, ki so enakovredne nalogam Gradle razširitve Java. Tarča *clean* izbriše vsebino imenika *build*. Tarča *assemble* zgradi zložno kodo, vendar moramo sami zagotoviti, da se vse potrebne knjižnice nahajajo v imeniku *dependencies*. Tarča *jar* zgradi arhiv JAR, vendar se mora pred tem uspešno izvesti tarča *assemble*, kar definiramo z lastnostjo *depends*. Opazimo

lahko, da je uporaba ogrodja Gradle precej enostavnejša, saj splošno funkcionalnost enostavno dodamo z uporabo razširitev, ki jih nato lahko prilagodimo potrebam projekta.

3.4.2 Primerjava z orodjem Maven

Orodje Maven je bilo izdano leta 2004 in je poskušalo odpraviti pomanjkljivosti orodja Ant. Omogočalo je naprednejše upravljanje z knjižnicami, pri čemer te lahko tudi prenese iz oddaljenih odlagališč (*repository*). Tako kot orodje Ant tudi Maven uporablja format XML za opis nastavitev, ki jih hrani v datoteki *pom.xml*. Namesto tarč ta uporablja definiran življenjski cikel, ki je sestavljen iz več faz. Te si sledijo v predpisanem vrstnem redu in že vsebujejo potrebno funkcionalnost za gradnjo Java projektov. Vsebovana funkcionalnost je vsebinsko podobna Gradle razširitvi Java, katero predstavljajo faze *clean*, *compile*, *build*, *package*, *install* in *deploy*. Te so natančno definirane, kljub temu jim funkcionalnost lahko dodajamo z razširitvami.

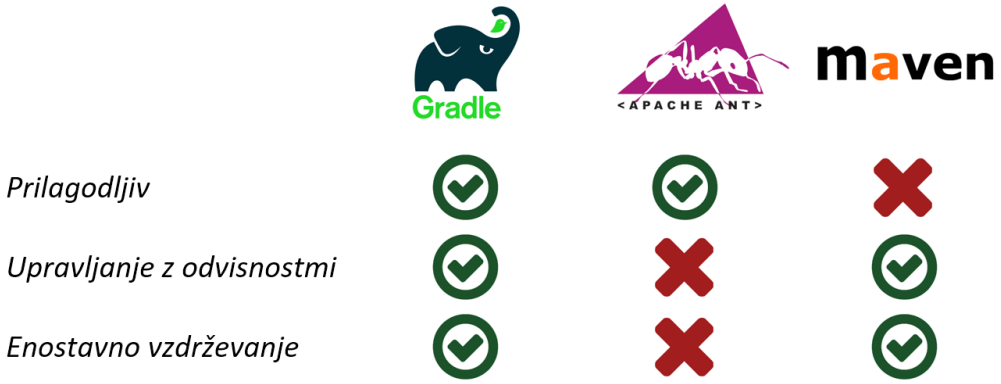
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="..." xsi:schemaLocation="...">
   <groupId>si.fri.demo.is</groupId>
   <artifactId>ejb</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>ejb</packaging>
   <properties>
      <maven.compiler.target>1.8</maven.compiler.target>
   </properties>
   <dependencies>
      <dependency>
         <groupId>javax</groupId>
         <artifactId>javaee-api</artifactId>
         <version>7.0</version>
      </dependency>
   </dependencies>
   <build>
      <plugins>
         <plugin>
            <artifactId>maven-ear-plugin</artifactId>
              <version>2.10.1</version>
              <configuration> ... </configuration>
         </plugin>
      </plugins>
   </build>
</project>
```

Odsek 3.8: Vsebina datoteke *pom.xml* za gradnjo z orodjem Maven

Odsek 3.8 prikazuje vsebino *pom.xml* datoteke, ki je vsebinsko enaka že predstavljenemu Gradle projektu. Edino razliko predstavlja dodatek *maven-ear-plugin*, ki omogoči gradnjo EAR arhiva namesto JAR. To lahko v Gradle projektu storimo z razširitvijo Ear, ki ustvari novo nalogo in ne spreminja starih.

3.4.3 Sklep

Gradle tako uporablja najboljše lastnosti obeh orodji in odpravlja njuni pomanjkljivosti, kar prikažemo na sliki 3.1. Hkrati velja tudi za hitrejše orodje, kot njuna predhodnika Ant in Maven. To so razvijalci dosegli z uporabo zalednega procesa, ki se vzpostavi ob prvem zagonu in se kasneje ponovno uporablja. Ponovno nalaganje in zagon navideznega stroja JVM zato ni potrebno, kar pospeši izvajanje gradnje projekta na račun večje porabe pomnilnika. V zadnjih letih se je njegova uporaba zelo razširila in je hkrati postal de facto standard za gradnjo Android projektov. Pričelo ga je uporabljati tudi podjetje Netflix, ki ponuja velik nabor odprtokodnih rešitev, med katerimi je tudi Gradle razširitev Nebula.



Slika 3.1: Primerjava orodji Gradle, Maven in Ant

Poglavje 4

Zahteve informacijskega sistema

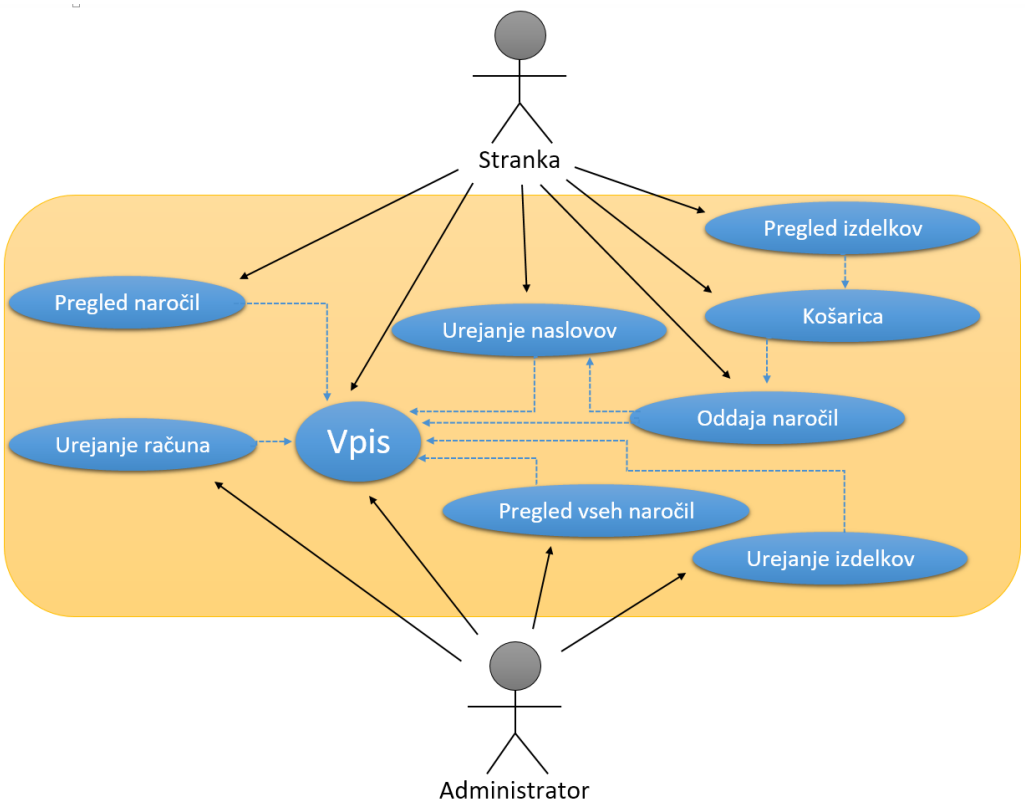
V diplomski nalogi bomo predstavili razvoj enostavnega informacijskega sistema trgovine. Implementirali bomo le osnovne funkcionalnosti, saj bomo večji poudarek dali sami strukturi sistema, splošnosti zasnove in uporabljenim tehnologijam, tako da bomo postavili solidne temelje za nadaljnji razvoj.

V uvodnem poglavju smo že na kratko omenili zahteve našega informacijskega sistema, ki jih bomo v tem poglavju podrobneje predstavili. Razdelili smo jih v dve skupini. Vsebinske zahteve določajo vse potrebne funkcionalnosti sistema za stranke in administratorje, ki jih predstavlja UML diagram na sliki 4.1. Infrastrukturne zahteve podrobno opisujejo potrebne zmogljivosti in lastnosti sistema.

4.1 Vsebinske zahteve

Informacijski sistem spletne trgovine bo vseboval glavni nabor funkcionalnosti današnjih konkurenčnih spletnih trgovin, kot so Amazon, EBay, Alibaba, itd. Za uporabnike, ki so v vlogi strank, mora informacijski sistem tako implementirati sledeče funkcionalnosti:

- pregled izdelkov - stranka lahko brez vpisa v sistem pregleduje izdelke,



Slika 4.1: Vsebinske zahteve prikazane z UML diagramom

ki jih lahko tudi filtrira. Ker so rezultati poizvedb lahko obsežni, je potrebno omogočiti tudi paginacijo ter razvrščanje,

- hranjenje izdelkov v košarici - ne prijavljena stranka lahko hrani izdelke, ki jih namerava kupiti, v košarici. Ta lahko vsebuje poljubno število izdelkov s količino, ki jih stranka lahko poljubno spreminja vse do oddaje naročila,

- registracija in vpis v sistem - stranki moramo omogočiti ustvarjanje novega uporabniškega računa v vlogi stranke, ki je potreben za oddajo naročil. Za ustvaritev novega računa mora stranka posredovati elektronski poštni naslov, ki ga je potrebno preveriti. Ko je veljavnost elektronskega naslova potrjena, je stranki dovoljen vpis. Sistem mora omogočati tudi ponastavitev gesla, če ga stranka pozabi,

- urejanje podatkov računa - vpisana stranka lahko spreminja podatke o svojem računu. Če spreminja elektronski naslov, mora tega ponovno potrditi,
- dodajanje in urejanje lastnih naslovov za dostavo - vpisana stranka si lahko lasti poljubno število naslovov za dostavo, ki jih lahko ureja,
- oddaja naročila - vpisana stranka lahko ustvari naročilo z uporabo košarice, ki mu mora pred oddajo izbrati naslov za dostavo,
- pregled naročil - vpisana stranka lahko pregleduje lastna že oddana naročila,

Poleg strank sistem potrebuje tudi administratorje. Ti predstavljajo zaposlene, ki nadzorujejo in upravljajo spletno trgovino. Za opravljanje svojega dela ti potrebujejo naslednje funkcionalnosti:

- vpis in registracija v sistem - nov uporabniški račun v vlogi administratorja za zaposlene lahko ustvari le administrator sistema,
- urejanje podatkov računa - vpisani administrator lahko ureja le podatke svojega računa,
- dodajanje in urejanje izdelkov - vpisani administrator lahko dodaja, briše in ureja prodajne izdelke. To vključuje nastavljanje cene in popusta,
- pregled vseh oddanih naročil - prijavljen administrator lahko pregleduje vsa oddana naročila strank,

4.1.1 Hranjenje vsebine

Za realizacijo opisanih zahtev je potrebno hraniti podatke o izdelkih, strankah, naslovih, naročilih in administratorjih. Za posamezen izdelek potrebujemo podatek o nazivu, opisu, ceni in popustu. Za stranko in administratorja

moramo hraniti podatke o poštnem naslovu, imenu, priimku, geslu in telefonski številki. Pri naslovu hranimo poštno številko, pošto, naslov ter stranko, ki si ga lasti. Za naročilo moramo poleg stranke hraniti tudi naslov za dostavo ter vse izbrane izdelke s količino. Tu moramo paziti, da spreminjanje izdelkov ne vpliva na že oddana naročila. Enako pravilo velja tudi za naslov dostave, ki se na naročilu ne sme spreminjati.

4.2 Infrastrukturne zahteve

Informacijski sistem mora poleg že omenjenih vsebinskih zahtev zadostiti tudi sledečim infrastrukturnim zahtevam:

- prenosljivost - informacijski sistem mora biti neodvisen od operacijskih sistemov,
- fleksibilnost - dodajanje novih tehnologij oz. menjava obstoječih, ki niso več ustrezne, mora biti enostavno in brez večjih posegov v obstoječo arhitekturo. Uporabljene tehnologije naj bodo odprtokodne, če je to mogoče,
- skalabilnost - ob povečanju prometa se mora informacijski sistem ustrezno skalirati,
- varnost - vzpostavljena povezava med odjemalcem in strežnikom mora biti varna. Zaznati je potrebno napade z grobo silo ter varno shranjevati uporabniška gesla,
- dokumentacija - funkcionalnosti informacijskega sistema in njihova uporaba mora biti dobro dokumentirana za lažjo integracijo v druge sisteme,
- cena - prednost imajo brezplačne oz. cenovno ugodnejše tehnologije, ki ustrezno rešujejo zadani problem,

Poglavje 5

Uporabljene tehnologije

V tem poglavju podrobneje predstavimo izbrane tehnologije, ki jih bomo uporabili pri razvoju informacijskega sistema. Izbrali smo jih na podlagi infrastrukturnih zahtev, zato so vse neodvisne od operacijskega sistema, odprtokodne in brezplačne. Hkrati njihova medsebojna integracija nudi ustrezen nivo fleksibilnosti in skalabilnosti.

5.1 Platforma Java EE

Platforma Java EE [18] se uporablja tudi za razvoj spletnih storitev. Njena specifikacija vsebuje velik nabor tehnologij, s katerimi enostavno implementiramo obsežno funkcionalnost. Implementacijo spletnih storitev izvedemo v objektnem programskem jeziku Java, ki je bila razvita na podlagi sledečih principov [27]:

- Enostavno, objektno orientirano
- Zanesljivost in varnost
- Arhitekturna neodvisnost in prenosljivost
- Zmogljivost

Programski jezik Java tvori zložno kodo, ki za izvajanje potrebuje navidezni stroj JVM. To poslabša hitrost izvajanja aplikacije, vendar nudi tudi nekaj prednosti. Javanska zložna koda se brez ponovne prevedbe lahko izvaja na različnih napravah in operacijskih sistemih. Hkrati razvijalcem ni potrebno skrbeti za pomnilnik tekom razvoja, saj se ta samodejno sprošča z uporabo Garbage Collector-ja, ki je del navideznega storja JVM. Med izvrševanjem lahko tudi pregledujemo izvorno kodo (*Reflections*), kar nam nudi veliko fleksibilnosti pri generični zasnovi. Java je hitro postala priljubljen programski jezik, zaradi omenjenih lastnosti. Stran PYPL [31], ki ocenjuje popularnost programskih jezikov, Javo postavlja na prvo mesto z 22,7% deležem. Temu primerna je tudi obsežnost uporabniške baza, kar je glavni razlog za veliko število knjižnic ter orodij. Za pričetek razvoja potrebujemo namestiti JDK, medtem ko za izvajanje aplikacij potrebujemo le JRE. V nadaljevanju bomo predstavili tehnologije platforme Java EE, ki jih potrebujemo za realizacijo zalednega dela informacijskega sistema. Poslovne aplikacije zgrajene s tehnologijami platforme Java EE za izvajanje potrebujejo aplikacijski strežnik, ki podpira vse uporabljene tehnologije.

5.1.1 JAX-RS

Del platforme Java EE je tehnologija JAX-RS, ki se uporablja za razvoj REST vmesnikov. Vmesnik bi lahko razvili z uporabo servletov, vendar bi morali implementirati veliko splošne funkcionalnosti, kot je vsebinsko pogajanje (*content negotiation*) in upravljanje z varnostnimi vlogami, kar bi podaljšalo čas razvoja in otežilo vzdrževanje izvorne kode. Če želimo uporabiti tehnologijo JAX-RS 2.0, moramo njegovo podporo zagotoviti na izbranem aplikacijskem strežniku.

Tehnologija ponuja bogat nabor anotacij s katerimi pregledno in enostavno implementiramo REST vmesnik. Ključne anotacije za implementacijo so sledeče:

- @ApplicationPath - predstavlja začetno točko vmesnika in nosi vrednost predpone poti prikazano v odseku 5.1,

- @Path - anotacija vsebuje vrednost predpone poti in se uporablja na razredu ali metodi,
- @GET, @POST, @PUT, @DELETE - anotacije se uporabljajo na metodi, ki se bo odzivala na zahtevke z anotirano metodo
- @PathParam - uporablja se pri definiciji argumentov metode, katere pridobi iz poti
- @QueryParam - uporablja se pri definiciji argumentov metode, katere pridobi iz parametrov poizvedbe
- @Consumes - definira podprte formate vsebine v zahtevku,
- @Produces - definira podprte formate vsebine v odgovoru,

```
@ApplicationPath("/api")
public class Api extends Application { ... }
```

Odsek 5.1: Primer začetne točke REST vmesnik s predpono poti */api*

```
@RequestScoped
@Path("Product")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ProductResource {

  @GET
  @Path("{id}")
  public Response get(@PathParam("id") Integer id) { ... }

  @GET
  public Response get(@QueryParam("id") Integer id) { ... }

  @POST
  public Response get(Product object) { ... }

}
```

Odsek 5.2: Primer Java razreda z JAX-RS anotacijami, ki predstavlja vir izdelka

Enostaven primer vira izdelkov prikazuje odsek 5.2, ki vsebuje tri operacije. Prva operacija se odziva na zahtevek z metodo GET in potjo */api/Product/1* iz katere pridobi parameter *id*. Druga operacija parameter *id*

pridobi iz parametra poizvedbe GET zahtevka s potjo */api/Product?id=1*. Tretja operacija se odziva na POST zahtevek s potjo */api/Product* in v telesu pričakuje izdelek v formatu JSON.

5.1.2 JPA

Informacijski sistem podatke shranjuje v relacijsko podatkovno bazo. Za lažjo integracijo baze bomo uporabili ORM imenovan JPA, ki nam preslika Java razrede v tabele relacijske podatkovne baze. Ta tehnologija izpostavi vmesnik za komunikacijo z bazo preko razreda EntityManager. Razred vsebuje metode *persist*, *find*, *merge* in *remove*, ki predstavljajo operacije CRUD. Tehnologija JPA je prenosljiva med podatkovnimi bazami. Razvijalci so to dosegli z uporabo vmesnega SQL jezika imenovanega JPQL, ki je po strukturi podoben preostalim SQL jezikom. Ta se pretvori v SQL jezik želene podatkovne baze s pomočjo JDBC gonilnika. Menjava podatkovne baze zato ni problematična, saj potrebuje le ustrezen JDBC gonilnik. Tabele v bazi predstavimo z razredi v Javi, ki jih moramo ustrezno anotirati. Primer takega razreda vsebuje odsek 5.3. Anotacija @Entity označuje da gre za razred, ki predstavlja tabelo s stolpci *id*, *title* in *price*. Možno je definirati tudi relacijo med tabelama z anotacijo @OneToMany in @ManyToOne.

```
@Entity
@Table(name="product")
public class Product {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Column(name = "id")
    protected Integer id;

    @Column(length =128, nullable = false)
    private String title;

    @Column(precision = 8, scale = 4, nullable = false)
    private BigDecimal price;
}
```

Odsek 5.3: Primer Java razreda z JPA anotacijami, ki predstavlja tabelo izdelkov

Da tehnologija JPA prepozna prave razrede, je to potrebno navesti v konfiguracijski datoteki *./META-INF/persistance.xml*. Poleg razredov moramo

v njej definirati podatkovni vir, ki ga ponuja izvajalno okolje. Za hitrejši razvoj tehnologija JPA omogoča tudi ustvarjanje podatkovne sheme direktno v sami bazi.

5.1.3 EJB

Poslovno logiko, ki vsebuje pravila o shranjevanju in spreminjanju podatkov, bomo implementirali v poslovnih zrnih s tehnologijo EJB. Ta nam omogoča modularno gradnjo logike, ki jo nato poljubno integriramo s tehnologijo kot je JAX-RS. EJB definira naslednje anotacije za razvoj različnih tipov poslovnih zrn:

- @Stateless - ne ohranja stanja seje,
- @Statefull - podatki se ohranjajo za posamezne seje, na katere se odjemalci sklicujejo z uporabo piškotkov,
- @Singleton - poslovno zrno se ustvari le enkrat ob njegovi prvi uporabi in se v nadaljevanju ponovno uporablja v novih sejah,

Poslovno zrno @Statless ponuja najboljšo zmogljivost, saj ne potrebuje pomnilnika za hranjenje stanja sej. @Stateful zrna so še vedno zmogljiva, če zagotovimo ustrezno velikost pomnilnika za željno število uporabnikov. Temu ni tako v poslovnih zrnih @Singleton. Da v teh ne prihaja do konfliktov branja in pisanja, je razred sinhroniziran, kar omogoča dostop le eni izvajalni niti hkrati. Uporaba teh tipov zrn se zato odsvetuje, saj predstavljajo ozko grlo, kadar do njih dostopa veliko število sej.

Poslovna zrna implementirajo tudi transakcije in varnost, ki jo enostavno prilagajamo posameznim varnostnim vlogam z anotacijama @PermitAll in @RolesAllowed. Primer takega zrna vidimo v odseku 5.4, ki vsebuje anotacije @Statless, @PermitAll, @Local, @PersistenceContext in @PostConstruct. @Stateless definira tip poslovnega zrna brez hranjenja stanj. @Local definira vmesnik DatabaseServiceLocal, preko katerega dostopamo do zrna z

uporabo anotacije @EJB. Hkrati njegovo uporabo dovolimo vsem uporabnikom z anotacijo @PermitAll. Metode anotirane s @PostConstruct se izvedejo po vzpostaviti zrna, kar omogoča uporabo vstavljenih zrn ali tehnologij. V prikazanem primeru je uporabljena anotacija @PersistenceContext, ki vstavi razred EntityManager tehnologije JPA, preko katerega manipuliramo podatkovno bazo.

```
@PermitAll
@Stateless
@Local(DatabaseServiceLocal.class)
public class DatabaseService implements DatabaseServiceLocal {

    @PersistenceContext(unitName = "is-jpa")
    private EntityManager em;

    private Database database;

    @PostConstruct
    private void init(){
        this.database = new Database(em);
    }...
}
{...
@EJB
DatabaseServiceLocal database
...}
```

Odsek 5.4: Primer Java razreda z EJB anotacijami, ki predstavlja poslovno zrno

5.2 Gradle razširitve

Podrobneje smo že predstavili Gradle razširitev Java, vendar ta ni dovolj za potrebe našega informacijskega sistema, zato smo uporabili še dodatne razširitve:

- War - razširitev War dopolnjuje razširitev Java, od katere je tudi odvisna. Uporablja se za gradnjo spletnih arhivov. Ta poleg zložne kode razredov vsebuje tudi odvisnosti oz. knjižnice ter nastavitveno datoteko *./WEB-INF/web.xml*, ki je potrebna za namestitev na aplikacijski strežnik,
- Ear - Tako kot razširitev War tudi razširitev Ear dopolnjuje razširitev Java. Omogoča gradnjo poslovnih arhivov, ki zložno kodo deli na

odvisnosti oz. knjižnice ter module. Definira tudi vsebino datoteke *./META-INF/application.xml*. V njej so opisane nastavitve varnostnih vlog, modulov in knjižnic, ki so potrebne za namestitev na aplikacijski strežnik.

- Dependency-management - Za lažje usklajevanje med različicami odvisnosti bomo uporabili razširitev Dependency-management. Vse uporabljene odvisnosti in njene različice bomo definirali v korenskem modulu ter se na njih sklicevali v preostalih modulih. Odvisnosti lahko definiramo samostojno ali kot del večje skupine. S tem zagotovimo enotno verzijo odvisnosti po celotnem projektu, kar prepreči konflikt združljivosti različnih verzij odvisnosti med moduli.

- Liquibase - Že omenjena tehnologija JPA omogoča gradnjo tabel v podatkovnih bazah, vendar ta ne nudi podpore migracij. To funkcionalnost dodamo z razširitvijo Liquibase. Spremembe strukture podatkovne baze hranimo v datoteki *changelogs.groovy*, ki jo v nadaljevanju razvoja dopolnjujemo. Najmanjšo spremembo opišemo z lastnostjo *changeSet*, ki mora imeti unikatno lastnost *id*. Preko te lastnosti Liquibase ve, za katero spremembo gre in ali je ta že bila izvedena nad podatkovno bazo. Podatke stanja baze razširitev Liquibase hrani v tabeli *databasechangelog*, ki se ustvari ob prvi uporabi. Posamezne spremembe opišemo z DSL-om orodja Liquibase, ki jih lahko tudi razveljavimo, če te niso ustrezne.

- Cargo - Zgrajeno EAR aplikacijo moramo namestiti na aplikacijski strežnik. Ročno nameščanje v razvojnem okolju je lahko zelo zamudno, kadar moramo to opravljati pogosto. Ta problem rešimo z uporabo razširitve Cargo, ki avtomatizira zagon strežnika in namestitev aplikacije. To lahko izvede na oddaljenem ali lokalnem strežniku, ki ga lahko tudi samodejno namesti. Razširitev je združljiva z velikim naborom aplikacijskih strežnikov, zato menjavo tega lahko izvedemo brez večjih sprememb nastavitev razširitve.

5.3 Podatkovna baza PostgreSQL

Izbira relacijske podatkovne baze ni preprosto opravilo. Vsaka baza ima namreč svoje prednosti in slabosti, zato se te izberejo na podlagi zahtev projekta. Eno izmed dobrih načel izbire je tudi razširjenost uporabe in podpora tako ponudnika kot skupnosti. Po analizi strani db-engines [8] med najbolj razširjene sodijo Oracle DB, MySQL, Microsoft SQL Server, PostgreSQL in MongoDB. Za uporabo baze Oracla DB potrebujemo licenco, zato njena uporaba v našem primeru odprtokodne rešitve ni primerna. Enako velja za bazi Microsoft SQL Server, ki ni prenosljiva na druge operacijske sisteme in MongoDB, ki za shranjevanje podatkov namesto tabel uporablja dokumente shranjene v formatu BSON. Izbiro skrčimo na bazi MySQL in PostgreSQL. Podatkovni bazi sta si funkcionalno zelo podobni, kljub temu baza PosgreSQL podpira obsežnejšo funkcionalnost. Vsebuje večje število naprednih podatkovnih tipov ter hkrati uporabniku omogoča definicijo novih. Podpira ustvarjanje materialnih pogledov (*materialized view*), s katerimi pospeši poizvedbe na račun prostora na disku. Z uporabo okenskih funkcij (*window functions*) in pogostih tabelnih izrazov (*common table expressions*) lahko tudi zgradimo naprednejše poizvedbe. Poizvedba se lahko izvaja tudi paralelno, vendar mora biti ta temu prilagojena. To so glavni razlogi za izbor podatkovne baze PostgreSQL 9.6 [30]. Po potrebi lahko podatkovno bazo kasneje tudi zamenjamo s katerokoli podatkovno bazo, ki ima svojo implementacijo JDBC gonilnika.

5.4 Avtentikacijski strežnik Keycloak

Del informacijskega sistema je tudi avtentikacijski strežnik, preko katerega poteka vpis uporabnikov. Uporabili bomo strežnik ponudnika JBoss imenovan Keycloak 3.1.0.Final [20]. Z njim enostavno integriramo bogato funkcionalnost za upravljanje z uporabniki in njihovimi vlogami. Hkrati že vsebuje vpisno okno, preko katerega uporabnikom omogočimo ustvarjanje računa in vpis v informacijski sistem. Za ustvaritev novega računa uporabnik potre-

buje veljaven naslov elektronske pošte. Tega strežnik po potrebi tudi preveri s pošiljanjem enkratne povezave, ki jo uporabnik obišče in s tem potrdi veljavnost poštnega naslova ter aktivira račun. Uporabnikom lahko dovolimo tudi vpis preko ponudnikov identitet, kot sta Google in Facebook.

Keycloak lahko gosti poljubno število področji (*realm*), ki vsebujejo svoje nastavitve, odjemalce, varnostne vloge in uporabnike. Kljub temu se odsvetuje uporabo velikega števila področji, saj to poslabša zmogljivost strežnika, zato je te bolje razdeliti na več posameznih strežnikov. Varnost je pri strežniku ključnega pomena, zato ta že vsebuje funkcionalnost za zaznavanje in preprečevanje napadov z grobo silo. V primeru vdora v podatkovno bazo strežnik gesla hrani zakodirana z algoritmom PBKDF2, kar prepreči njihovo zlorabo.

5.4.1 Protokol OAuth 2.0

REST vmesnik bomo zavarovali z uporabo protokola OAuth 2.0. Za vpis poleg elektronskega poštnega naslova in gesla uporabnika potrebujemo spletni naslov strežnika, ime področja ter odjemalca. Področju odjemalca ustvarimo v nadzorni plošči in mu definiramo veljavne povratne naslove. Vpis uporabnika pričnemo s preusmeritvijo na avtentikacijski strežnik. Pri tem podamo naslov za povratno preusmeritev in ime odjemalca v parametrih poizvedbe. Povratni naslov se mora ujemati z veljavnimi naslovi odjemalca, saj je v nasprotnem primeru vpis zavrnjen. Po uspešni prijavi v sistem strežnik uporabnika preusmeri na podani povratni naslov in pri tem vključi vpisno kodo v parametru poizvedbe. To vpisno kodo nato uporabimo za pridobitev dostopnega in obnovitvenega žetona. Dostopni žeton dodamo vsakemu zahtevku v glavo z lastnostjo *Authorization*. Ker so žetoni podpisani z asimetričnim algoritmom RS256, aplikacijski strežnik s pridobljenim javnim ključem Keycloak strežnika sam potrdi veljavnost žetona. V žetonu se nahajajo vsi potrebni podatki, kot so varnostne vloge, identifikacijski UUID uporabnika, datum preteka, itd. Z vključitvijo podatkov o vpisanem uporabniku v vsakem zahtevku hranjenje sej ni potrebno, kar ustreza načelu REST tehnologije.

5.5 Aplikacijski strežnik Wildfly

Razvita poslovna aplikacija s tehnologijami platforme Java EE se zapakira v arhiv WAR ali EAR, ki se moramo za izvajanje namestiti na ustrezen aplikacijski strežnik. Danes je na trgu mnogo različnih ponudnikov aplikacijskih strežnikov, kot so JBoss (Wildfly), Oracle (GlassFish, Weblogic), IBM (Websphere), Apache (TomEE). Podprtost Java EE 7 specifikacije se med njimi razlikuje. Potrebno je natančno pregledati dokumentacije strežnikov ter podprtost posameznih tehnologij, ki jih potrebujemo. Med izbranimi smo se odločili za ponudnika JBoss in njihov aplikacijski strežnik Wildfly [21]. Uporaba tega je brezplačna pod pogoji licence GNU Lesser General Public License v2.1. Hkrati podpira tudi celoten nabor Java EE 7 specifikacije, nudi obsežno dokumentacijo in omogoča enostavno integracijo z avtentikacijskim strežnikom Keycloak.

Strežnik Wildfly je možno zagnati v spletnem ali polnem profilu. Spletni profil vsebuje le osnovne tehnologije platforme Java EE za izvajanje spletnih strani z uporabo servletov. Nastavitve spletnega profila se nahajajo v datoteki *standalone.xml*, ki se privzeto uporabi ob zagonu. Ta nabor tehnologij ni zadosten za naš projekt, zato bomo uporabili poln profil, ki vsebuje celoten nabor tehnologij platforme Java EE. Ob zagonu moramo zato uporabiti datoteko *standalone-full.xml*.

5.6 Ogrodje KumuluzEE

V začetku bomo razvili EAR aplikacijo za aplikacijski strežnik. Taka arhitektura aplikacije je monolitna, saj vse funkcionalnosti vsebuje v eni datoteki. Ta pristop se že dolgo uporablja in ima bogat nabor razvojnih okolij. Kljub temu se je v zadnjih letih s pojavom oblačnih storitev razširila uporaba mikrostoritev. Mikrostoritve funkcionalnosti razdelijo na manjše celote, ki se izvajajo neodvisno druga od druge [24]. Taka arhitektura nam omogoča enostavno skaliranje posameznih funkcionalnosti. Kadar skaliramo monolitne aplikacije z namestitvijo novih strežnikov, moramo ponovno vključiti celotno

aplikacijo vključno z aplikacijskim strežnikom. To povzroči veliko porabo pomnilnika, kar rešuje arhitektura mikrostoritev, ki omogoča skaliranje posameznih funkcionalnosti. Pri tem moramo upoštevati, da je večina aplikacij danes še vedno monolitnih, vendar je pričakovati, da bodo podjetja iz obstoječe monolitne arhitekture postopoma prenašala del funkcionalnosti na mikrostoritve.

V nadaljevanju bomo implementirali mikrostoritev, ki bo obdelala del zahtevkov namenjenih poizvedbi izdelkov in s tem razbremenila aplikacijski strežnik. Za implementacijo mikrostoritve bomo uporabili tehnologije platforme Java EE. Za izvajanje bomo namesto aplikacijskega strežnika uproabili ogrodje KumuluzEE 2.2.0 [23], ki s svojo modularno zgradbo ponuja velik nivo fleksibilnosti. Jedro ogrodja KumuluzEE ne vsebuje nikakršne tehnologije Java EE platforme, zato je te potrebno dodati kot odvisnosti. S tem lahko drastično zmanjša končno velikost aplikacije, saj vključimo le tehnologij, ki jih naša aplikacija potrebuje.

5.7 Ogrodje React

Implementacija uporabniškega vmesnika, ki v ozadju komunicira z REST vmesnikom predstavlja velik izziv, saj morajo ti ohranjati in zagotavljati pravilno stanje aplikacije. REST tehnologija ne ohranja stanja med sejami, zato vse podatke pridobi iz zahtevka. Ti morajo vsebovati vse potrebne podatke za izvršitev operacije, saj pošiljanje v delih ni mogoče. REST vmesnik za komunikacijo uporablja protokol HTTP, zato lahko uporabniški vmesnik implementiramo neodvisno od naprave, operacijskega sistema in programskega jezika. Kot smo že omenili upravljanje s stanjem uporabniškega vmesnika hitro postane težko obvladljivo in z implementacijo več različnih vmesnikov obvladljivost samo še poslabšamo. Temu se izognemo z ustrezno izbiro tehnologije vmesnika, ki se lahko izvaja na obsežnem naboru naprav. Omenjenemu kriteriju ustreza tehnologija JavaScript, ki se izvaja v vseh sodobnih brskalnikih na večini naprav.

Klasično spletno stran strežnik ustvari ob vsakem zahtevku, ki jo nato prikaže brskalnik. Večina dela tu opravlja strežnik, ki ohranja tudi podatke seje za posameznega uporabnika. Ta način se slabo skalira, predvsem zaradi obsežne porabe procesnih virov. Da bi se izognili tej oviri, je bil razvit nov tip spletnih strani. Ta se obnaša kot aplikacija, ki se v celoti prenese ob prvem obisku spletne strani in se nato hrani v predpomnilniku brskalnika. Aplikacija s tehnologijo JavaScript implementira vso potrebno logiko za nadaljnje spreminjanje strani in komunikacijo z zalednim sistemom. Zgled in obnašanje aplikacije sta na prvi pogled enaka klasičnemu pristopu, vendar je takoj razvidno, da se odzivnost aplikacija precej izboljša. Spletni strani ni potrebno čakati na odgovor strežnika ter jo ponovno nalagati, ker za izgled skrbi logika na strani odjemalca. S tem pristopom velik del procesiranja prestavimo s strežnika na odjemalca.

Za enostavnejši razvoj take aplikacije uporabimo ogrodje React [12], čigar priprava projekta za razvoj aplikacije je zelo obsežna, saj imamo na voljo veliko dodatnih tehnologij in knjižnic, s katerimi pohitrimo razvoj. Uporabili bomo pripravljen projekt za razvoj naprednih aplikacij react-boilerplate [1]. Ta poleg ogrodja React vsebuje tudi ostale knjižnice kot sta Redux, ki omogoča pregledno shrambo stanja aplikacije, in react-router, s katerim organiziramo poti. Za hitrejši razvoj ima že pripravljen strežnik Express v načinu hot module reload, ki samodejno ponovno naloži spremenjene datoteke. Omeniti moramo še orodje webpack, ki združi vse JavaScript datoteke v eno ter jo minimizira in s tem drastično pospeši nalaganje strani v brskalnikih.

5.8 Obratni namestniški strežnik Nginx

Naš informacijski sistem tako sestoji iz aplikacijskega strežnika, mikrostoritve, avtentikacijskega strežnika ter strežnika za serviranje JavaScript aplikacije. Uporabniki bodo do storitev dostopali preko spletnega brskalnika v katerem se bo izvajala JavaScript aplikacija. Ta se bo prenesla iz strežnika

Express in bo po vzpostavitvi pričela uporabljati avtentikacijski strežnik, aplikacijski strežnik in mikrostoritev. Vsi strežniki se lahko izvajajo na isti strojni opremi in so dostopni na istem naslovu, vendar uporabljajo različna vrata (*port*). Današni brskalniki privzeto preprečujejo komunikacijo s tujimi viri, zato bi morali te ekplicitno dovoliti z uporabmo mehanizma CORS. Temu se izognemo z uporabo obratnega namestniškega strežnika Nginx 1.12.0 [25], ki omogoča dostop do vseh strežnikov preko enotnega vira, kar izboljša preglednost. Strežnik Nginx opravlja tudi vlogo razporejevalnika zahtevkov med aplikacijskimi strežniki in mikrostoritvami, kar omogoča skaliranje sistema. Uporabljamo ga tudi za predpomnjenje HTTP odgovorov na zahtevke z metodo GET, kar razbremeni in poveča odzivnost sistema.

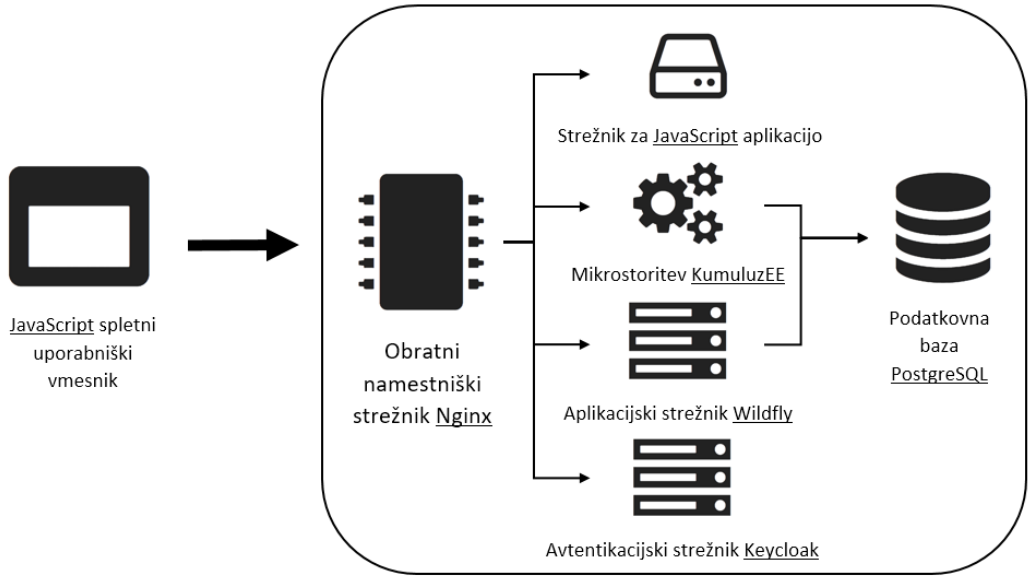
Poglavje 6

Razvoj informacijskega sistema

V tem poglavju predstavimo razvoj informacijskega sistema s pomočjo tehnologij, katerih uporabo smo na podlagi podanih specifikacij upravičili v prejšnjem poglavju. Poglavje je sestavljeno iz dveh delov. V prvem delu smo zasnovali strukturo projekta in implementirali glavno funkcionalnost informacijskega sistema po pristopu bottom-up, ki smo jo zapakirali v arhiv EAR. Hkrati razvijemo mikrostoritev z ogrodjem KumuluzEE. Zasnovali smo tudi spletni uporabniški vmesnik v ogrodju Rect in Java API knjižnico za komunikacijo z REST vmesnikom. V drugem delu se osredotočimo na namestitev potrebne infrastrukture za izvajanje EAR aplikacije in mikrostoritve. Sprva namestimo in ustrezno nastavimo podatkovno bazo PostgreSQL in avtentikacijski strežnik Keycloak. Temu bo sledila namestitev aplikacijskega strežnika Wildfly. Dodamo še mikrostoritev, ter strežnik Express za strežbo JavaScript aplikacije. Nato vse skupaj postavimo za obratni namestniški strežnik Nginx. Tako strukturo informacijskega sistema prikazuje slika 6.1.

6.1 Implementacija funkcionalnosti

V začetku zasnove informacijskega sistema moramo dobro poznati njegove zahteve, saj tako lahko ustrezno prilagodimo strukturo projekta. Ta mora biti hkrati tudi prilagodljiva na kasnejše spremembe zahtev, zato projekt



Slika 6.1: Zasnovana infrastruktura informacijskega sistema.

razdelimo na vsebinske celote imenovane modul. Ta predstavlja zaključeno celoto funkcionalnosti dela projekta, ki vsebuje odvisnosti na knjižnice ali druge module v projektu. Razširjen način delitve REST Java EE projekta je na module *shema baze*, *poslovna zrna*, *REST vmesnik* in *arhiv poslovne aplikacije EAR*.

```
is (informacijski sistem)
|
+-- core (jedro)
|   +-- jpa (shema baze)
|   +-- businessLogic (poslovna logika)
|   +-- restComponents (REST komponente)
+-- app (Aplikacija)
|   +-- microservice (mikrostoritev)
|   |   +-- rest (REST vmesnik)
|   +-- server (streznik)
|       +-- ejb (poslovna zrna)
|       +-- rest (REST vmesnik)
|       +-- ear (arhiv poslovne aplikacije EAR)
+-- api (vmesnik)
+-- client (odjemalec)
    +-- web (splet vmesnik)
```

Odsek 6.1: Modularna struktura Gradle projekta s prevodi

Omenjene module smo v diplomskem projektu ohranili, vendar smo jih še nadaljnje razdelili. Strukturo projekta predstavlja odsek 6.1 s korenskim

imenikom *informacijski sistem*, ki vsebuje imenike *jedro*, *aplikacija*, *vmesnik* in *odjemalec*. Ker želimo izbrani projekt graditi z orodjem Gradle, moramo vsakemu modulu ustvariti datoteko *build.gradle*. Ta vsebuje podatke o verziji, odvisnostih, postopku gradnje ter načinu testiranja posameznega modula. Datoteko *build.gradle* vsebuje tudi korenski imenik v katerem hranimo skupne lastnosti, ki jih dedujejo preostali moduli. Med te lastnosti sodita razširitvi Java in Dependency-management. V korenu imamo tudi datoteko *settings.gradle* z vsebino odseka 6.2, ki definira relativne poti modulov. Modul *strežnik* tudi vsebuje datoteko *build.gradle*, vendar le z navodili za testiranje EAR aplikacije na aplikacijskem strežniku Wildfly.

```
rootProject.name = 'is'
include 'core:jpa'
include 'core:businessLogic'
include 'core:restComponents'
include 'app:server'
include 'app:server:ejb'
include 'app:server:rest'
include 'app:server:ear'
include 'app:microservice'
include 'api'
```

Odsek 6.2: Vsebina datoteke *settings.gradle*, ki definira relativne poti korenskih imenikov posameznih modulov

6.1.1 Jedro

Modul *jedro* vsebuje module *shema baze*, *poslovna logika* in *REST komponente*, ki predstavljajo osrednjo funkcionalnost projekta. Na te module se bodo v nadaljevanju sklicevali moduli *strežnik*, *vmesnik* in *mikrostoritev*.

Shema podatkovne baze

Za razvoj podatkovne sheme smo uporabili pristop Code-First. Tabele predstavimo z razredi, iz katerih nato ustvarimo shemo v podatkovni bazi. Razrede anotiramo z uporabo anotacij knjižnice javaee-api, ki jo je potrebno dodati med odvisnosti. Razvoj pričnemo z abstraktnim razredom BaseEntity prikazan v odseku 6.3. V njem hranimo skupne lastnosti tabel, ki za pravilno prepoznavanje potrebuje anotacijo @MappedSuperclass.

```
@MappedSuperclass
public abstract class BaseEntity<T extends BaseEntity>
implements Serializable {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    @Column(name = "id")
    protected Integer id;

    @Version
    protected Integer version;

    @Column(name = "is_deleted", nullable = false)
    protected Boolean isDeleted;

    @Column(name = "created_on", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    protected Date createdOn;

    @Column(name = "edited_on", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    protected Date editedOn;
    ...
}
```

Odsek 6.3: Abstraktni razred BaseEntity

Vse tabele imajo tako sledeče skupne lastnosti:

- Lastnost *id* oz. identifikator je predstavljena s celim številom med 1 in 2^{32} in enolično določa vrstico v tabeli. Podatkovna baza bo skrbela za dodeljevanje novih identifikatorjev, ki so za 1 večji od predhodnikov.

- Lastnost *version* anotirana z @Version hrani število izvedenih sprememb, kar omogoča zaznavanje zastarelih različic. Kadar do iste vrstice dostopa več odjemalcev in jo vsi tudi spreminjajo se pojavi problem zagotavljanja usklajenosti podatkov. Če nek odjemalec spremeni vrstico se različica poveča za ena in drugemu odjemalcu onemogoči posodabljanje, dokler ta ne pridobi najnovejše različice.

- Lastnost *isDeleted* določa ali gre za izbrisano entiteto, ki jo še vedno hranimo v podatkovni bazi. V relacijski bazi vrstice ne moremo enostavno izbrisati, kadar se nanjo sklicuje neka druga vrstice s tujim ključem. V takih primerih je uporaba označitve za izbrisano primernejša, kar tudi omogoča obnovo že izbrisanih entitet.

- Lastnost *createdOn* vsebuje datum in uro, ko je bila vrstica ustvarjena.

- Lastnost *editedOn* vsebuje datum in uro, ko je bila vrstica nazadnje spremenjena.

Poleg razreda BaseEntity smo implementirali tudi abstraktni razred BaseEntityVersion prikazan v odseku 6.4. Ta deduje iz razreda BaseEntity ter doda svoje lastnosti in metode, s katerimi verzioniramo entiteto, kar omogoča pregledovanje zgodovine sprememb. Razred BaseEntityVersion doda naslednje lastnosti:

- Lastnost *originId* hrani identifikator, ki se dodeli prvi verziji, ko je bila ta ustvarjena.
- Lastnost *versionOrder* hrani številko verzije, ki se ob vsaki spremembi poveča za ena.
- Lastnost *isLatest* pove ali je različica najnovejša.

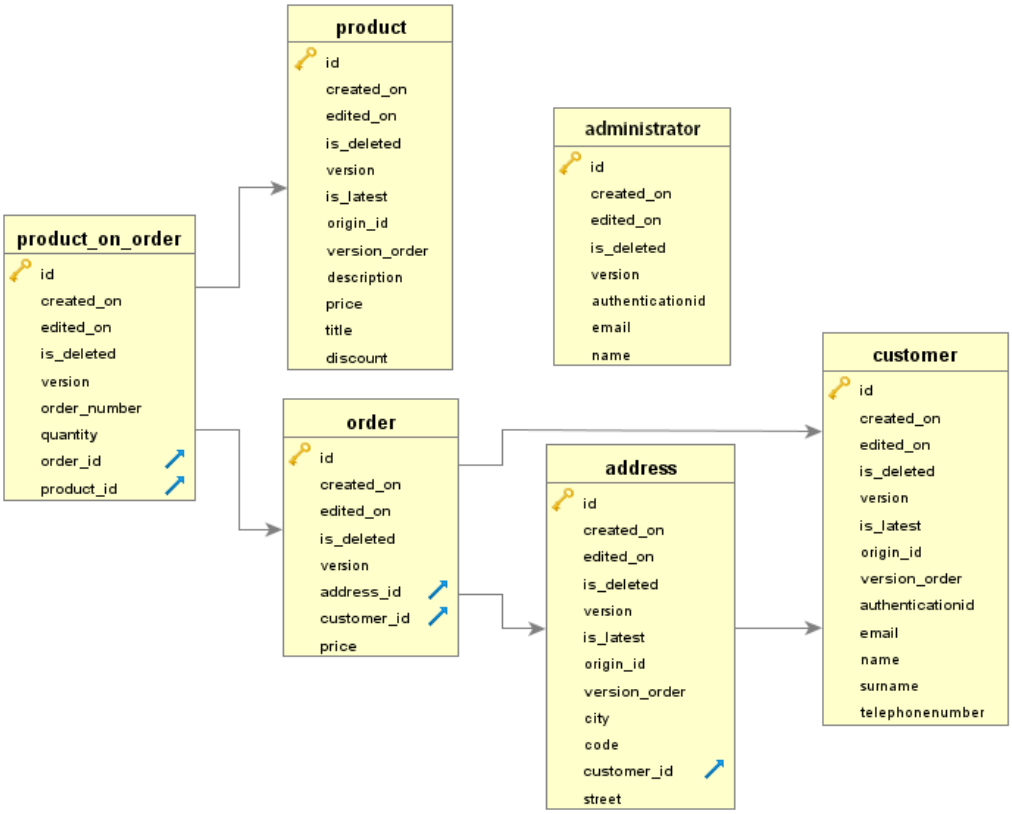
V splošnem podatke v bazi prepišemo z novimi, medtem ko pri vodenju verzij vedno ustvarimo novo vrstico, ki hrani podatke zadnje verzije. Delovanje bomo natančneje predstavili v modulu *poslovna logika*.

```
@MappedSuperclass
public abstract class BaseEntityVersion
<T extends BaseEntityVersion> extends BaseEntity<T> {
    @Column(name = "origin_id")
    protected Integer originId;

    @Column(name = "version_order", nullable = false)
    protected Integer versionOrder;

    @Column(name = "is_latest", nullable = false)
    protected Boolean isLatest;
    ...
}
```

Odsek 6.4: Abstraktni razred BaseEntityVersion

Podatkovna shema vsebuje šest tabel, ki jih predstavimo z razredi, ki dedujejo iz ustreznega abstraktnega razreda. Informacijski sistem omogoča pregled izdelkov, katerih podatke bomo hranili v tabeli *product*. Tabela vsebuje *ime*, *opis*, *ceno* in *popust* posameznega izdelka. Te lastnosti se lahko spreminjajo, zato je potrebno hraniti zgodovino že prodanih izdelkov. Razred Product zato deduje iz abstraktnega razreda BaseEntityVersion. Naročilo

stranka odda s poljubnimi številom izdelkov, zato potrebujemo dve novi tabeli. Prva tabela imenovana *order* bo hranila podatke naročil. Druga tabela imenovana *product_on_order* bo povezovala izdelke z naročilom in hranila podatke o količini. Naročilo je vezano na stranko, zato dodamo še tabelo imenovano *customer*. Poleg stranke naročilo potrebuje tudi naslov, ki ga hranimo v tabeli *address* in pripada posamezni stranki. Potrebno je dodati tudi tabelo za hranjenje podatkov administratorjev imenovano *administrator*. Končni zgled podatkovne sheme prikazuje slika 6.2.



Slika 6.2: Podatkovna shema informacijskega sistema

Poslovna logika

V tem modulu hranimo ključno poslovno logiko, ki za osnovo uporablja modul *shema baze*. Glavni del predstavlja razred Database, ki nosi logiko za

ustvarjanje, pridobivanje, spreminjanje in brisanje entitet v podatkovni bazi. Služi kot fasada tehnologiji JPA z uporabo razreda EntityManager, ki ga uporablja za manipulacijo podatkovne baze. Razred EntityManager vsebuje metode *find*, *persist*, *merge* in *remove*, ki kot parameter sprejmejo objekt preslikan v tabelo. Da bi preprečili vnos nepravilnega tipa objekta, v ovojnih metodah fasadnega razreda Database uvedemo omejitev, ki zahteva dedovanje iz razreda BaseEnity ali BaseEnityVersion, če uporabljamo metode namenjene verzioniranju.

Funkcionalnost GET zahtevka implementiramo s pomočjo metode *find*, ki za parametra sprejme tip razreda ter identifikator. Kot rezultat nam metoda vrne entiteto, če ta obstaja. Če podani identifikator ne ustreza nobeni entiteti, metoda vrne prazen rezultat, zato vržemo napako s statusom NOT FOUND. Poleg pridobivanja ene entitete z uporabo metode *find*, lahko tabele podatkovne baze poizvedujemo tudi po drugih lastnostih. Tehnologija JPA uporablja svojo različico SQL jezika imenovan JPQL, s katerim lahko uporabimo tudi bolj kompleksne poizvedbe. Uporabniki podajajo filtre za iskanje preko parametrov poizvedb, ki se nato prevedejo v poizvedbo JPQL. To funkcionalnost vsebuje knjižnica lynx v razredih QueryParameters in JPAUtils. Razred QueryParameters iz URL naslova izlušči filtre in njihove vrednosti. Te nato podamo razredu JPAUtils, ki ustvari in izvede SQL poizvedbo.

Entitete ustvarimo z metodo *create* razreda Database, ki zapisuje v bazo z uporabo metode *persist*. Pred zapisom v bazo preverimo če je struktura entitete ustrezna, ima vpisan uporabnik ustrezne pravice ter nastavimo datuma nastanka in spremembe. Kadar ustvarjamo entiteto, ki deduje iz razreda BaseEntityVersion, moramo nastaviti tudi izvirni identifikator, številko verzije ter indikator najnovejše različice.

Entitete spreminjamo na dva načina z uporabo metode *update*, ki predstavlja zahtevek PUT ali metodo *patch*, ki predstavlja zahtevek PATCH. Pri metodi *update* moramo posredovati celotno entiteto, s katero bomo prepisali staro. Temu ni tako pri metodi *patch*, saj moramo tam podati le vrednosti lastnosti, ki jih spreminjamo. Ta pristop zmanjša velikost zahtevka, kar vodi v

boljšo odzivnost sistema. Entitete, pri katerih vodimo verzije, posodabljamo na podoben način. Razlika je v pisanju podatkov v bazo. Privzeti način spremembe zapiše v obstoječo entiteto in tako izgubimo prejšnje vrednosti. Da se izognemo izgubi podatkov, ustvarimo novo entiteto, kar poteka v dveh korakih. Začnemo s prepisovanjem vseh vrednosti iz predhodne verzije, ki jih nato v drugem koraku spremenimo s podanimi vrednostmi zahtevka. Pri tem moramo nastaviti tudi številko nove verzije, ki je za ena večja od prejšnje ter jo označiti kot najnovejšo. Kasneje za pridobitev najnovejših verzij posameznih entitet uporabimo poizvedbo s pogojem *isLatest=true*, katero s parametri poizvedbe predstavimo kot *where=isLatest:eq:true*.

Entitete izbrišemo z metodo *permDelete*, ki uporablja metodo *remove* razreda EntityManager. Brisanje entitet v nekaterih primerih ni zaželeno, zato namesto brisanja uporabljamo metodo *delete*, ki entiteto z oznako *isDeleted* le označi kot izbrisano. Entitete označene kot izbrisane lahko filtriramo in jih ne prikazujemo v poizvedbah. Hkrati jo lahko obnovimo, če pride do napake pri brisanju.

Metodam razreda Database lahko tudi podamo logiko za preverjanja lastništva in preverjanja ustreznosti podatkov. Kadar stranka ustvari naslov ali odda naročilo nastavimo ustrezno lastništvo. Lastništvo je treba preverjati tudi pri pregledovanju ali spreminjanju entitet. Stranka lahko namreč pregleduje le naslove in naročila, ki si jih lasti.

Poleg razreda Database modul *poslovne logike* vsebuje tudi upravitelje, ki vsebujejo dodatno logiko. Upravitelja uporablja razred naročil, saj privzeta logika POST zahtevka ni ustrezna za shranjevanje naročil, ki imajo strogo predpisano obliko. Upravitelj naročil preveri veljavnost naročila, ki mora vsebovati vsaj en izdelek ali več, ki imajo veljaven identifikator in količino večjo od nič. Podajanje stranke v zahtevku ni nujno, saj se ta pridobi iz varnostnega konteksta. Upravitelj preveri tudi lastništvo podanega naslova, ki mora ustrezati vpisani stranki. Če v naročilu ni napake, se to shrani v podatkovno bazo. Poleg razreda naročil upravitelja uporabljata tudi razreda strank in administratorjev. Stranka ali administrator ustvari nov račun

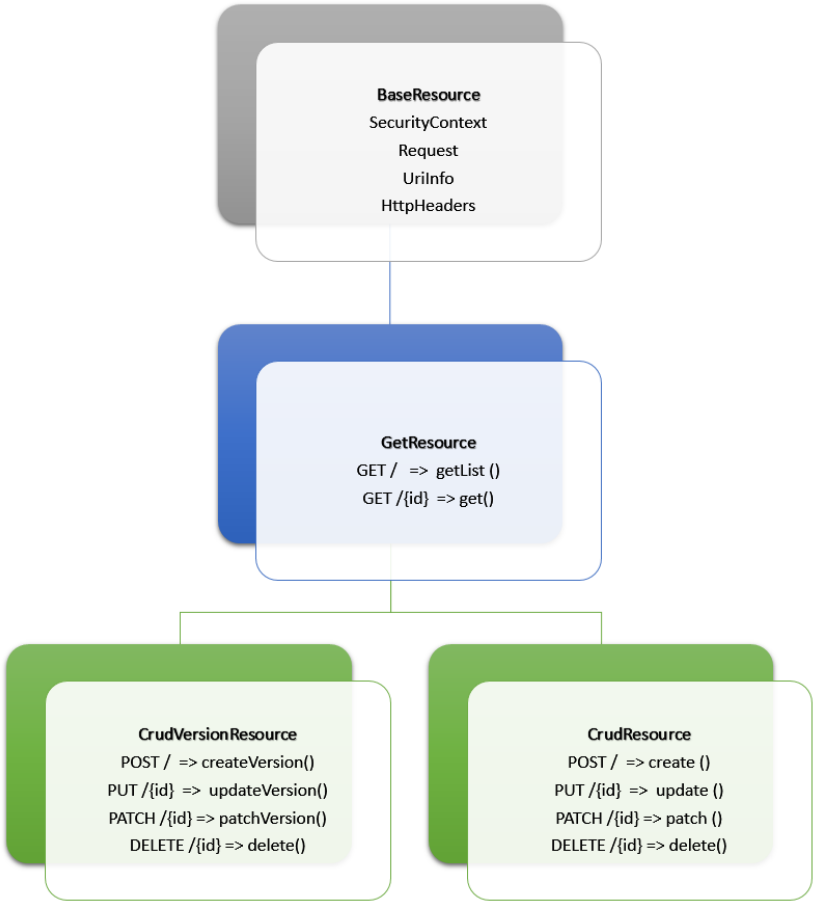
na avtentikacijskem strežniku, katerega podatkovna baza ni del glavne podatkovne baze. Račun se v glavni podatkovni glavni bazi ustvari, ko prvič kličemo upravitelja.

REST komponente

Modul temelji na prejšnjem modulu poslovne logike in na njem gradi generične REST komponente. Modul vsebuje tri generične abstraktne razrede s katerimi gradimo REST vmesnik. Ti kot generični parameter sprejemajo tipe razredov, ki dedujejo iz razreda BaseEntity. Za podani tip ustvarijo vir, ki ga lahko nato ustrezno prilagodimo. Implementacijo začnemo z abstraktnim razredom BaseResource, ki vsebuje kontekste, kot so *varnost*, *zahtevek*, *glave zahtevka* in *pot zahtevka*. Preko teh dostopamo do podatkov o posameznem zahtevku, kar nam omogoča pridobitev podatkov o vpisanem uporabniku in paramatrih poizvedbe.

Abstraktni razred GetResource deduje iz razreda BaseResource in vsebuje metodo *get* in *getList*, ki se odzivata na HTTP GET zahtevek. Metoda *getList* z razredom QueryParameters iz konteksta poti zahtevka pridobi filtre in jih poda metodi *getList* razreda Database. Ta vrne seznam entitet in njihovo celotno število, iz česar sestavimo odgovor. Entitete vrnemo kot seznam JSON objektov v telesu, medtem ko število vseh entitet podamo v glavi z vrednostjo v lastnosti *X-Count*. Metoda *get* iz poti zahtevka pridobi parameter *id*, ki predstavlja identifikator entitete. Entiteto s podanim identifikatorjem nato pridobi z metode *get* v razredu Database in jo vrne skupaj z glavo *ETag*.

Abstraktni razred CrudResource deduje iz razreda GetResource in poleg podedovanih metod *getList* in *get* vsebuje tudi *create*, *update*, *patch* in *delete*. Te metode skupaj predstavljajo CRUD operacije iz česar izhaja tudi ime razreda. Metoda *create* obdeluje POST zahtevke. Entiteto v telesu zahtevka posreduje metodi *create* razreda Database, ki jo shrani v bazo. Če se entiteta uspešno ustvari vrne odgovor s statusom CREATED in potjo do ustvarjene entitete v glavi *Location*. Metodi *update* in *patch* pričakujeta



Slika 6.3: Prikaz generične zasnove abstraktnih razredov za implementacijo virov REST vmesnika

entiteto v telesu ter identifikator v poti zahtevka. Vsebino zahtevka nato posredujeta enakovrednima metodama v razredu Database. V splošnem metodi ne vračata spremenjene entitete, saj so nam spremembe znane v zahtevku. Njun odgovor ima prazno telo s statusom No Content, kar ni vedno zaželen način obravnave zahtevka. Vrednosti polja, kot je datum spremembe, ki ga nastavlja strežnik iz zahtevka ne poznamo in moramo v takem primeru ponovno poslati GET zahtevek. Zato smo v našem projektu podprli dva načina obravnave zahtevka s pomočjo lastnosti v glavi imenovano *X-Content*. Kadar je vrednost lastnosti true, vmesnik vrne celotno entiteto v telesu odgovora.

Vir entitet z vodenjem verzij implementiramo z abstraktnim razredom CrudVersionResource. Ta je funkcijsko podoben razredu CrudResource, vendar kot generični parameter sprejme le tipe razredov, ki dedujejo iz razreda BaseVersionEntity. Razlika je tudi v klicanju metod razreda Database, ki namesto metod *update*, *create* in *patch* uporablja metode *updateVersion*, *createVersion* in *patchVersion*. Te metode ustvarjajo nove različice entitet, katere smo podrobneje predstavili v modulu *poslovna logika*.

Končano implementacijo virov prikazuje slika 6.3, kjer je dedovanje prikazano s povezavo. Implikacija predstavlja metodo razreda Database, ki se kličejo pri obdelavi zahtevka. Taka zasnova razredov nam omogoča hitro implementacijo novih virov in hkrati nudi fleksibilnost prilagajanja potrebam posameznih entitet.

6.1.2 Strežnik

Modul strežnik vsebuje module *poslovna zrna*, *REST vmesnik*, *arhiv poslovne aplikacije EAR* in testne razrede. Kljub temu le modula *poslovna zrna* in *REST vmesnik* vsebujeta izvorno kodo, medtem ko se modul *arhiv poslovne aplikacije EAR* uporablja zgolj za gradnjo poslovnega arhiva EAR, ki vsebuje celotno funkcionalnost poslovne aplikacije.

Poslovna zrna

Enterprise JavaBeans oz. poslovna zrna vsebujejo poslovno logiko aplikacije, ki smo jo že implementirali v modulu *poslovna logika*. Ponovna implementacija zato ni potrebna, saj modul dodamo kot odvisnost. Poslovna zrna bodo imela vlogo fasade razredom poslovne logike in pri tem preverjala pravice uporabnikov in omogočala uporabo transakcij. Potrebno je tudi implementirati vmesnike poslovnih zrn, ki izpostavijo metode za njihovo uporabo v drugih poslovnih zrnih ali servlet-ih. Kadar te potrebujemo, do njih dostopamo z anotacijo @EJB prikazano v odseku 6.5. Strežnik ob dobljenem zahtevku ustvari potrebna zrna, ki se razlikujejo po tipu. V uvodnih poglavjih smo predstavili načela tehnologije REST, ki nasprotujejo ohranjanju

stanj seje, zato bomo uporabili zrna @Stateless.

```
@Stateless
@Local(DatabaseServiceLocal.class)
public class DatabaseService implements DatabaseServiceLocal { ... }

{ ...
@EJB
protected DatabaseServiceLocal databaseService;
... }
```

Odsek 6.5: Definicija poslovnega zrna baze in njegova uporaba z anotacijo @EJB

Ustvarimo tudi varnostni vlogi stranka in administrator. Vloga stranke dovoljuje pregledovanje izdelkov, urejanje naslovov in oddajanje naročil, medtem ko vloga administratorja dovoljuje urejanje izdelkov in pregledovanje naročil. Vlogo vpisanega uporabnika pridobimo v žetonu JWT, ki ga izda avtentikacijski strežnik Keycloak. Posamezni vlogi dovoljujemo uporabo metode z anotacijo @RolesAllowed. Dovoljenja posameznih vlog bomo natančneje definirali v modulu *REST vmesnik*, kjer imamo boljši nadzor nad posamezno operacijo. Modul *poslovna zrna* tako vsebuj sledeča poslovna zrna:

- DatabaseService - fasada logike baze (Database), katere uporaba je dovoljena vsem uporabnikom,
- CustomerService - fasada logike upravljavca strank (CustomerManager), katere uporaba je dovoljena uporabnikom v vlogi stranke,
- OrderService - fasada logike upravljavca naročil (OrderManager), katere uporaba je dovoljena uporabnikom v vlogi stranke,
- AdministratorService - fasada logike upravljavca administratorjev (AdministratorManager), katere uporaba je dovoljena uporabnikom v vlogi administratorja,

REST vmesnik

Modul vsebuje končne implementacije posameznih virov, ki jih zgradimo z uporabo generičnih komponent v modulu *REST komponente*. Modul smo

zapakirali v arhiv spletne aplikacije, kar storimo z Gradle nalogo *war*, ki je del razširitve War. Implementacijo REST vmesnik pričnemo z razredom ISApplicationV1, ki deduje iz razreda Application in je anotiran z @ApplicationPath. Ta predstavlja začetno točko vmesnika in v anotaciji vsebuje predpono poti */api/v1/*. Model baze vsebuje šest tabel, vendar za povezovalno tabelo med izdelki in naročilom vira ne potrebujemo, zato vmesnik sestavljajo naslednji viri:

- ProductResource deduje iz CrudVersionResource, ki je dostopen na */Product*
- AddressResource deduje iz CrudVersionResource, ki je dostopen na */Address*
- OrderResource deduje iz GetResource, ki je dostopen na */Order*
- CustomerResource deduje iz CrudVersionResource, ki je dostopen na */Customer*
- AdministratorResource deduje iz CrudResource, ki je dostopen na */Administrator*

Vsak vir predstavlja tabelo v podatkovni bazi in nudi svoj nabor operacij, s katerimi manipuliramo predstavljeno tabelo. Do posameznega vira dostopamo s potjo oblike */api/v1/*{*ime vira*}, kjer {*ime vira*} nadomestimo z imenom želenega vira. Če želimo dostopati do vira izdelkov uporabimo pot */api/v1/Product*.

Izdelki morajo ohranjati zgodovino, zato njihov razred deduje iz razreda BaseEntityVersion. Potrebno je ustvariti tudi vir, ki uporablja metode razreda Database namenjene verzionizaciji. Vir izdelkov, ki ga predstavlja razred ProductResource, zato deduje iz abstraktnega razreda CrudVersionResource. Razred ProductResource ni abstrakten in mora implementirati vse abstraktne metoda iz dedovanega razreda. Abstraktni razred CrudVersionResource vsebuje dve abstraktni metodi *getDatabaseService* in *getAutho-*

rizedEntity. Metoda *getDatabaseService* vrača objekt, ki implementira vmesnik DatabaseImpl. Ta vmesnik pripada razredu Database in izpostavlja vse njegove metode. Vmesnik implementira tudi poslovno zrno DatabaseService, do katerega v razredu ProductResource dostopamo z anotacijo @EJB ter ga vrnemo v odgovoru metode *getDatabaseService*. Implementirati moramo tudi metodo *getAuthorizedEntity*, ki vrne objekt AuthEntity. Ta objekt vsebuje podatke vpisanega uporabnika. Ustvarimo ga z uporabo varnostnega konteksta, ki ob klicu metode *getUserPrincipal* vrne podatke o vpisanem uporabniku. Ta del funkcionalnosti vira izdelkov prikazuje izvorna koda v odseku 6.6.

```
@Path("Product")
@RequestScoped
public class ProductResource extends CrudVersionResource<Product> {

    @EJB
    private DatabaseServiceLocal databaseImpl;

    protected DatabaseImpl getDatabaseService() {
        return databaseImpl;
    }

    protected AuthEntity getAuthorizedEntity() {
        return AuthUtility.getAuthorizedEntity(
        (KeycloakPrincipal) sc.getUserPrincipal());
    }
    ...
}
```

Odsek 6.6: Razred ProductResource, ki predstavlja vir izdelkov, z implementiranima metodama *getDatabaseService* in *getAuthorizedEntity*

V prejšnjem delu smo ustvarili tudi vlogi stranke in administratorja. Dovolili smo jima uporabo posameznih poslovnih zrn, vendar ta delitev ni dovolj. Vir izdelkov omogoča CRUD operacija nad tabelo, vendar vse operacije niso na voljo vsakemu uporabniku. Metodi *get* in *getList* predstavljata operaciji tipa READ, ki sta na viru izdelkov dovoljena vsem uporabnikom, in ju zato anotiramo s @PermitAll. Uporabo preostalih metod, ki omogočajo urejanje izdelkov, moramo splošnim uporabnikom prepovedati, zato jih anotiramo z @RolesAllowed(ROLE_ADMINISTRATOR). S tem omejimo njihovo uporabo na uporabnike, ki so v vlogi administratorja.

CRUD operacije z verzioniranjem dodamo tudi viroma strank in naslovov. Tako kot izdelki, se morajo podatki o stranki in naslovu ohranjati. Na njih se sklicujemo na naročilu in se zato ne smejo spreminjati. Naslovi pripadajo posamezni stranki, zato moramo pred kakršno koli operacijo preveriti njihovo lastništvo. To funkcionalnost implementiramo v metodi *initAuthorizationManager* razreda AddressResource. Metoda vrne nov razred AuthorizationManager, ki predstavlja upravljavca lastništva. Ta v konstruktorju sprejme objekt AuthEntity, ki jo pridobimo z metodo *getAuthorizedEntity*. Upravljavcu lastništva ob nastanku ponovno implementiramo metodo *checkAuthority*, ki vsebuje logiko za preverjanje lastništva. Ta je v temu primeru enostavna, saj preveri samo, če se vpisani uporabnik ujema z uporabnikom, ki si lasti naslov. V nasprotnem primeru vrnemo napako s statusom FORBIDDEN. Upravljavec lastništva poleg metode *checkAuthority* vsebuje tudi metodi *setAuthorityFilter* in *setEntityAuthority*. Metoda *setAuthorityFilter* se uporablja ob klicu funkcije *getList* in nastavi potreben filter, tako da vpisanemu uporabniku vrne le lastne naslove, medtem ko metoda *setEntityAuthority* nastavi lastništvo naslova, kadar ustvarimo novo entiteto. Podobno preverjanje lastništva imamo tudi pri viru strank, saj moramo zagotoviti, da vpisani uporabnik lahko spreminja le svoj profil.

Vir strank poleg CRUD operacij vsebuje tudi metodo *loginUserInfo*, ki se odziva na HTTP GET zahtevek s potjo */api/v1/Customer/login*. Metoda vrne profil vpisane stranke. Klic te metode je nujen po vpisu v sistem, saj ustvaritev novega računa na avtentikacijskem strežniku zahteva tudi ustvaritev novega profila v tabeli strank. Ta se ustvari ob prvem klicu metode *loginUserInfo*, ki v nadaljevanju vrača obstoječi profil. Enak pristop uporablja tudi vir administratorjev. To funkcionalnost smo že implementirali v upravitelju strank in upravitelju administratorjev, ki ju sedaj vstavimo v ustrezni razred in uporabimo.

Vsi omenjeni viri omogočajo uporabo CRUD operacij. Izjema je le vir naročil, saj po njihovi oddaji teh nočemo spreminjati. Vir naročil predstavlja razred OrderResource in deduje iz abstraktnega razreda GetResource.

Ta vsebuje le metodi *get* in *getList*, ki omogočata branje tabele. Vir naročil tako ne omogoča manipulacijo tabele z uporabo CRUD operacij. Da kupcu omogočimo oddajo naročil, ustvarimo novo metodo, ki se odziva na HTTP POST zahtevke s potjo */api/v1/Order/process*, ki v telesu nosi podatke o naročilu. Metoda *process* nato posreduje naročilo v obdelavo upravitelju naročil. Hkrati moramo tako kot pri viru naslovov tudi v viru naročil definirati upravljavca lastništva, ki strankam omogoča le pregled lastnih naročil.

Za namestitev modula na aplikacijskem strežniku moramo projektu dodati datoteko *./webapp/WEB-INF/web.xml*. V njej definiramo varnostni vlogi kupca in administratorja. Ker uporabljamo strežnik Keycloak, dodamo še datoteko z nastavitvami *./webapp/WEB-INF/keycloak.json*. V datoteki *web.xml* moramo definirati še področje in odjemalca, ki se bosta uporabljala za vpis uporabnikov. Končna vsebina datoteke *web.xml* je prikazana v odseku 6.7.

```
<web-app ...>
        <context-param>
        <param-name>resteasy.scan</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.role.based.security</param-name>
        <param-value>true</param-value>
    </context-param>
    <login-config>
        <auth-method>KEYCLOAK</auth-method>
        <realm-name>is</realm-name>
    </login-config>
    <security-role>
        <role-name>CUSTOMER</role-name>
    </security-role>
    <security-role>
        <role-name>ADMINISTRATOR</role-name>
    </security-role>
</web-app>
```

Odsek 6.7: Vsebina datoteke *web.xml*, v kateri definiramo Keycloak področje (*realm*) ter varnostni vlogi *CUSTOMER* in *ADMINISTRATOR*

Dokumentacija

REST vmesniki so lahko zelo obsežni, kadar ponujajo obsežen nabor virov, ki podpirajo različne metode. Za pričetek uporabe vmesnika je potrebno natančno preučiti njihovo dokumentacijo, ki jo ustvarimo po specifikaciji Ope-

nAPI. Uporabimo knjižnico io.swagger.swagger-jaxrs, ki vsebuje anotacije s katerimi opišemo vmesnik. Podatke prikazane v korenu opisa definiramo z anotacijo @SwaggerDefinition, ki se nahaja v razredu ISApplicationV1. Operacije na viru opišemo, tako da posamezno metodo anotiramo z @ApiOperation in @ApiResponses. @ApiOperation vsebuje predstavitev in opis operacije, medtem ko @ApiResponses opisuje tipe odgovorov in njihovo vsebino. OpenAPI specifikacija dokumentira tudi potrebne argumente posameznih operacij, ki jih pridobi pri pregledovanju izvorne kode. Tak pristop ustvarjanja dokumentacije pohitri razvoj in zmanjša možnost napake.

Arhiv poslovne aplikacije EAR

Modul *arhiv poslovne aplikacije EAR* ne vsebuje izvorne kode ali testov in se izključno uporablja samo za gradnjo poslovnega aplikacijskega arhiva EAR. Modulu potrebno funkcionalnost dodamo z Gradle razširitvijo Ear. Module in knjižnice, ki jih vključimo v datoteko EAR definiramo v datoteki *build.gradle*. Arhiv EAR bo vseboval modula *REST komponente* in *poslovna zrna*, ki ju med odvisnostmi dodamo z nastavitvijo *deploy*. Dodati moramo tudi vse preostale knjižnice, ki jih modula potrebujeta za izvajanje. To naredimo z nastavitvijo *earlib*, ki rekurzivno vključi vse odvisnosti.

```
dependencies {
    earlib (project(path: ':app:server:rest',
    configuration: 'compile')){
        exclude group: 'javax', module: 'javaee-api'
        exclude group: 'org.keycloak', module: 'keycloak-core'
    }
    deploy project(path: ':app:server:ejb', configuration: 'archives')
    deploy project(path: ':app:server:rest', configuration: 'archives')
}
```

Odsek 6.8: Definicija odvisnosti modula *poslovni aplikacijski arhiv EAR*

Aplikacijski strežnik že vsebuje nekatere knjižnice, zato moramo te izključiti iz arhiva. Podroben opis vsebovanih knjižnic aplikacijskega strežnika najdemo v njegovi dokumentaciji. Med te sodita knjižnici javaee-api ter keycloak-core, ki ju izključimo z nastavitvijo *exclude*. Takšen opis odvisnosti je prikazana v odseku 6.8.

Za konec moramo definirati še datoteko *./META-INF/application.xml*. V njen nastavimo podatke o arhivu, modulih in varnostnih vlogah. Z uporabo Gradle razširitve *Ear* to naredimo v datoteki *build.gradle*. Vsebina datoteke je podrobneje prikazana v odseku 6.9. Orodje Gradle ima tako vse potrebne informacije za gradnjo EAR arhiva, katerega namestimo na aplikacijski strežnik.

```
ear {
    libDirName 'APP-INF/lib'
    deploymentDescriptor {
        version = "1"
        applicationName = "IS"
        initializeInOrder = true
        displayName = "IS Ear"
        description = "Demo project"
        module(project.ejbArchive, "ejb")
        webModule(project.warArchive, project.warContextRoot)
        securityRole "ADMINISTRATOR"
        securityRole "CUSTOMER"
    }
}
```

Odsek 6.9: Nastavitve Gradle razširitve Ear za ustvarjanje datoteke *application.xml*, ki je potrebna za namestitev arhiva EAR na aplikacijski strežnik

6.1.3 Mikrostoritev

Glavni razlog za ločitev poslovne logike in generičnih REST komponent na svoje module je njihova ponovna uporaba s pomočjo implementacije v obliki mikrostoritev. Modul *mikrostoritev* je poleg teh dveh modulov odvisen tudi od ogrodja KumuluzEE. Ogrodje podpira omejeno število tehnologij platforme Java EE, med katerimi so tudi JAX-RS, JPA in CDI, ki bo nadomestil EJB. S temi tehnologijami smo zasnovali aplikacijo za aplikacijski strežnik, zato lahko ponovno uporabimo že obstoječo implementacijo. Ogrodje KumuluzEE je zasnovano modularno. Njegovo jedro ne vsebuje omenjenih tehnologij, vendar te lahko dodamo kot odvisnosti prikazano v odseku 6.10. Temu ni tako pri aplikacijskemu strežniku Wildfly, saj ta vsebuje celoten nabor tehnologij platforme Java EE. Vseh ne uporabljamo pogosto, zato odvečne samo zasedajo pomnilnik. Ogrodje KumuluzEE z uporabljenimi tehnologijami zaseda le 25 MB, medtem ko aplikacijski strežnik Wildfly zaseda 260

MB. V produkcijskem okolju tako prihranimo pomnilnika za faktor deset, kar se v oblačni platformi pri večjem številu strežnikov precej pozna.

Implementacijo REST vmesnika mikrostoritve pričnemo z razredom ISApplicationV1Micro, ki je funkcijsko enak razredu ISApplicationV1. Mikrostoritev bo vsebovala le vir izdelkov, kateremu bomo podprli operaciji *get* in *getList* z uporabo abstraktnega razreda GetResource. Pri tem moramo ponovno implementirati metodi *getAuthorizedEntity* in *getDatabaseService*. Pridobitev seznama izdelkov ne zahteva prijave, zato funkciji *getAuthorizedEntity* ni potrebno ustvariti upravljavca lastništva. To ne velja za metodo *getDatabaseService*, saj mora ta ustvariti vmesnik baze. V EAR aplikaciji smo uporabili poslovno zrno, vendar za branje entitet ne potrebujemo transakcij, zato lahko razred Database ustvarimo v viru ProductResource. Ta kljub temu potrebuje razred EntityManager, do katerega dostopamo z anotacijo @PersistenceContext. Zato razred Database ustvarimo v metodi init, saj se ta, zaradi anotacije @PostConstruct, izvede po vstavljanju razreda EntityManager. Mikrostoritev lahko nato zaženemo z nalogo *runKumuluzEE*, ki prične izvajanje glavnega razreda com.kumuluz.ee.EeApplication s preostalimi razredi, nastavitvami in knjižnicami modula.

```
dependencies {
    compile project(':core:restComponents')

    compile group: "com.kumuluz.ee", name: 'kumuluzee-core'
    compile group: "com.kumuluz.ee", name: 'kumuluzee-servlet-jetty'
    compile group: "com.kumuluz.ee", name: 'kumuluzee-jax-rs-jersey'
    compile group: "com.kumuluz.ee", name: 'kumuluzee-jpa-hibernate'
    compile group: "com.kumuluz.ee", name: 'kumuluzee-cdi-weld'
    ...
}
```

Odsek 6.10: Odvisnosti modula *mikrostoritev*, s katerimi dodamo posamezno komponento ogrodja KumuluzEE

Zaradi ponovne uporabe modulov *poslovna logika* ter *REST komponent*, vso potrebno funkcionalnost mikrostoritve implementiramo v treh razredih. Projekti, kateri poslovno logiko hranijo ločeno od poslovnih zrnih, brez večjih problemov naredijo prestop iz monolitne arhitekture na mikrostoritve. Več dela je potrebno pri projektih, ki imajo tesno sklopljeno arhitekturo.

6.1.4 Spletni uporabniški vmesnik

Naš vzorčni spletni uporabniški vmesnik bo implementiral ključne funkcionalnosti, kot so pregled izdelkov, vpis uporabnika, dodajanje naslovov dostave ter oddajo naročil. Uporabili bomo knjižnico material-ui, ki nudi komponente v stilu Material [6]. Dodati moramo še Keycloak adapter, ki omogoča vpis uporabnika in pridobitev žetonov. Za komunikacijo z REST vmesnikom uporabljamo JavaScript metodo *fetch*, ki ji podamo potrebne glave in vsebino zahtevka. Ta se izvede asinhrono (*async*), zato je potrebno definirati funkcijo za obdelavo odgovora. Primer pridobitve izdelkov z metodo *fetch* je prikazan v odseku 6.11.

```
function buildHeaders() {
  const isHeaders = new Headers();
  isHeaders.append('Content-Type', 'application/json');
  isHeaders.append('Accept', 'application/json');
  isHeaders.append('Authorization', `Bearer ${authorizationToken}`);

  return isHeaders;
}

function getProducts(search, skip = 0) {
  const params = {
    where: `isLatest:eq:true|isDeleted:eq:false|title:likeic:%${search}%`,
    limit: LIMIT,
    skip: Number(skip),
  };

  const esc = encodeURIComponent;
  const query = Object.keys(params)
    .map((k) => `${esc(k)}=${esc(params[k])}`)
    .join('&');

  const isConfig = {
    method: 'GET',
    headers: buildHeaders(),
    mode: 'cors',
    cache: 'default' };

  return fetch(`${RESOURCE_PRODUCT}?${query}`, isConfig)
    .then(checkStatus)
    .then(parseJSON);
}
```

Odsek 6.11: JavaScript implementacija funkcionalnosti za pridobitev izdelkov iz razvitega REST vmesnika

Funkcija *getProducts* sprejme dva parametra. Prvi parameter *search* vsebuje besedo, katero primerjamo z naslovom izdelka. Drugi parameter *skip* predstavlja število izdelkov, ki jih preskočimo in se uporablja za paginacijo.

Filtre v zahteveku moramo tudi ustrezno zakodirati, kar naredimo z uporabo komponente *encodeURIComponent*. Ko zahtevek pridobi odgovor, se pokliče funkcija *checkStatus*, ki preveri status odgovora. Če je status veljaven, se pokliče funkcija *parseJSON*, ki vrne objekt z glavami in vsebino odgovora za nadaljnjo obdelavo. Za strežbo statične vsebine JavaScript datotek bomo uporabili strežnik Express, ki je že del pripravljenega projekta. Tega zaženemo z ukazom *npm start* in se izvaja na vratih (*port*) 3000.

6.2 Namestitev infrastrukture

V tem podpoglavju si bomo natančneje ogledali namestitev potrebnih tehnologij za izvajanje poslovne aplikacije. Tako kot razvoj projekta bo namestitev infrastrukture potekala po načelu bottom-up, zato namestitev začnemo pri podatkovni bazi PostgreSQL, ki predstavlja podatkovni vir poslovne aplikacije. Temu bo sledila namestitev avtentikacijskega strežnika Keycloak, na kateremu ustvarimo novo področje (*realm*). Nato namestimo aplikacijski strežnik Wildfly, ki ga je potrebno integrirati s podatkovno bazo in avtentikacijskem strežnikom. Hkrati vzpostavimo tudi mikrostoritev KumuluzEE, ki kot podatkovni vir tudi uporablja podatkovno bazo PostgreSQL. Zagnati moramo tudi strežnik Express za strežbo JavaScript aplikacije. Temu sledi namestitev in konfiguracija obratnega namestniškega strežnika, ki omogoči dostop do mikrostoritve, statičnega strežnika z spletnim vmesnikom, aplikacijskega strežnika in avtentikacijskega strežnika preko skupnega vira.

6.2.1 Podatkovna baza

Uporabljeno podatkovno bazo PostreSQL namestimo z namestitvenim čarovnikom pridobljenim na strani `https://www.postgresql.org/download/`. Med namestitvijo se ustvari administratorski račun *postgres*, s katerim bomo v nadaljevanju spreminjali bazo. Do podatkovne baze dostopamo z grafičnim uporabniškim vmesnikom pgAdmin. Z njim ustvarimo glavno bazo *is* ter testno bazo *is_test*. Za večjo varnost se priporoča ustvaritev novega uporabniškega

računa, ki mu omejimo uporabo operacij nad posamezno podatkovno bazo. Za demonstracijo projekta to ni potrebno, zato bomo uporabljali le administratorski uporabniški račun *postgres*. Poslovna aplikaciji bo uporabljala transakcije, katerih uporabo omogočimo v datoteki *postgresql.conf*. V njej lastnosti *max_prepared_transactions* spremenimo iz privzete vrednosti 0 na 64, ki predstavlja število dovoljenih sočasnih transakcij.

Shemo podatkovne baze opisano z Java razredi v bazi ustvarimo s tehnologijo JPA. V nastavitveni datoteki *./resource/persistance.xml* moramo med lastnosti dodati vrstico <property name="..."value="drop-and-create"/>. S tem bo tehnologija JPA ob vzpostavitvi aplikacije izbrisala vse tabele ter jih ponovno ustvarila. V razvojnem okolju s tem načinom enostavno spreminjamo podatkovno bazo. Ta način ni primeren za produkcijsko okolje, saj ne smemo izgubiti podatkov. Ta problem rešimo z uvedbo migracij, kar nam omogoči Gradle razširitev Liquibase.

Z njegovo uporabo pričnemo, ko imamo pripravljeno prvo verzijo baze za produkcijsko okolje. Tabele v bazi ustvarimo s tehnologijo JPA, nakar z Gradle nalogo *lbGenerateChangelog* ustvarimo opis vseh sprememb za ustvaritev začetnega stanja. Tega preimenujemo v *changelog_1.groovy* in ga shranimo v imenik *./resources/liquibase/changelogs*, v katerem bomo hranili vse nadaljnje opise sprememb. Na te opise se z oznako *include* sklicujemo v datoteki *changelogs.groovy*. Datoteko nato uporabljamo kot parameter, ki vsebuje opise vseh sprememb, ki se izvedejo ob klicu naloge *lbUpdate*. Pred dodajanjem novih sprememb moramo izvesti sinhronizacijo z začetnim opisom in bazo, kar naredimo z nalogo *lbChangelogSync*. Ta bo ustvarila dve novi tabeli imenovani *databsechangelog* in *databsechangeloglock*, ki se uporabljata za hranjenje izvedenih sprememb nad bazo. V razvojnih okoljih za testiranje delovanja potrebujemo tudi testne podatke, ki jih dodamo v datoteko *changelog-import.groovy* z oznako *sql*.

Dodajanje nadaljnjih sprememb moramo izvesti ročno. Pomagamo si lahko z nalogo *lbDiffChangeLog*. Ta kot parameter vzame bazi *is* in *is_test*, ki ju primerja med seboj in ustvari datoteko *changelog-diff.groovy*, ki vsebuje

vse potrebne spremembe za uskladitev baz. Spremembe sheme baze zato naredimo v Java razredih, iz katerih ponovno ustvarimo tabele v testni bazi *is_test* ter jih nato primerjamo s staro različico tabel v bazi *is*. S tem si zagotovimo usklajenost med ustvarjenim opisom sprememb in Java razredi. Ta pristop ima sicer nekaj omejitev pri preimenovanju tabel ali stolpcev. Ustvarjen opis sprememb bo izbrisal obstoječi stolpec in ga ponovno ustvaril z novim imenom, kar povzroči izgubo podatkov. V takem primeru moramo spremembe opisati ročno in jih testirati v razvojnem okolju.

V datoteki *build.gradle* smo za lažjo uporabo različnih baz in opisov sprememb definirali nalogi imenovani *mainDB* in *testDB*, ki ju uporabljamo za nastavljanje uporabljene baze. Če želimo ponovno zgradit podatkovno bazo *is* uporabimo, ukaz *gradle mainDB lbDropAll lbUpdate*. Če želimo v bazo vnesti testne podatke, moramo pri ukazu podati parameter *-Pcl=import*, ki zamenja privzeto datoteko opisa sprememb z *./changelog-improt.groovy*. Kadar hočemo spreminjati testno bazo, samo zamenjamo nalogo *mainDB* z *testDB*.

6.2.2 Avtentikacijski strežnik

Strežnik Keycloak prenesemo s strani `http://www.keycloak.org/downloads.html` in shranimo v želeni imenik. Izogniti se moramo problemu podvojenih vrat (*port*) z aplikacijskim strežnikom, zato pred prvim zagonom strežnika Keycloak v datoteki *standalone.xml* spremenimo lastnost *port-offse* v znački <socket-binding-group>. Privzeto vrednost 0 spremenimo na 1 in s tem vse vrata zamaknemo za vrednost 1. Privzeti vrata 8080 tako postanejo 8081. Strežnik nato poženemo z datoteko *./bin/standalone.bat* in obiščemo spletno stran na naslovu localhost:8081. Ob prvem obisku moramo ustvariti nov administratorski račun, s katerim se vpišemo v spletno konzolo. Privzeto je ustvarjeno samo področje (*realm*) *master*, ker želimo imeti uporabnike informacijskega sistema ločene od Keycloak administratorjev ustvarimo novo področje imenovano *is*. V njem definiramo varnostni vlogi za administratorje in stranke informacijskega sistema imenovani *CUSTOMER* in *ADMI-*

NISTRATOR. Če hočemo preverjati veljavnost elektronskih poštnih naslovov uporabnikov ob registraciji in dovoliti obnovo pozabljenih gesel, moramo področju *is* dodati še elektronski poštni račun s potrebnimi podatki za vpis.

Uporabnik ustvari nov račun z obiskom naslova `/auth/realms/is/account`. Ta predstavlja vpisno točko za področje *is*, na kateri se nahaja tudi povezava za obnovitev pozabljenega gesla ter ustvarjanje novega uporabniškega računa. Vpisani uporabnik lahko spreminja le podatke o svojem računu. Možno je spreminjanje naslova elektronske pošte, imena ter priimka. Po potrebi lahko dodamo tudi nove lastnosti. Za dodatno varnost uporabniku omogočimo uporabo dvostopenjskega preverjanja (*Two Factor Auth*) z Google Authenticator-jem. Ta se danes vedno pogosteje uporablja, predvsem v finančnih aplikacijah, kjer je to obvezno, zaradi vse bolj pogostih spletnih napadov.

6.2.3 Aplikacijski strežnik

Namestitev strežnika Wildfly 10.0.0 pričnemo s prenosom datoteke s strani `http://wildfly.org/`, ki jo po končanem prenosu shranimo v željen imenik. Strežnik nato zaženemo s skripto *./bin/standalone.bat* in obiščemo spletno stran na naslovu *localhost:8080*. Ob prvem obisku ustvarimo nov administratorski račun, ki ga uporabimo za prijavo v nadzorno konzolo na spletnem naslovu *localhost:9990*.

Ker EAR aplikacija uporablja tehnologijo JPA, je pred njeno namestitvijo potrebno dodati JDBC gonilnik za podatkovno bazo PostgresSQL. Tega prenesemo iz spletne strani `https://jdbc.postgresql.org/` in ga namestimo v imenik *./modules*. V datoteki *standalone-full.xml* moramo definirati še podatkovni vir. Definicijo pričnemo z značko <driver> s katero definiramo JDBC gonilnik. Nanj se nato sklicujemo v znački <xa-datasource>, ki predstavlja podatkovni vir s transakcijami. Značka mora vsebovati še naslov podatkovne baze, velikost bazena povezav ter uporabniško ime in geslo. Celotna definicija vira je prikazan v odseku 6.12. Za preverjanje vpisa uporabnika je potrebno namestiti tudi Keycloak adapter, ki ga prenesemo iz spletne strani

`http://www.keycloak.org/downloads.html`. Vsebino prenesenega arhiva shranimo v korenski imenik aplikacijskega strežnika Wildfly. Arhiv vsebuje imenik s potrebnimi moduli ter namestitveno skripto. Za izvršitev skripte z ukazom *./bin/jboss-cli.sh –connect –file="adapter-install.cli"* je potrebno zagnati strežnik.

```
...
<xa-datasource jndi-name="java:/jboss/datasources/ISDS" pool-name="ISDS"
enabled="true" spy="true">
        <xa-datasource-property name="Url">
                jdbc:postgresql://localhost:5432/is
        </xa-datasource-property>
        <driver>postgresql</driver>
        <xa-pool>
                <min-pool-size>1</min-pool-size>
                <max-pool-size>10</max-pool-size>
                <prefill>true</prefill>
        </xa-pool>
        <security>
                <user-name>postgres</user-name>
                <password>root</password>
        </security>
</xa-datasource>

...
<driver name="postgresql" module="org.postgresql">
        <xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-datasource-class>
        <datasource-class>org.postgresql.Driver</datasource-class>
</driver>
```

Odsek 6.12: Nastavitev podatkovnega vira za aplikacijski strežnik Wildfly

Skripta bo spremenila uporabljeno nastavitveno datoteko, zato je pomembno, da strežnik zaženemo z ukazom *./bin/standalone.bat -c standalone-full.xml* in uporabimo nastavitveno datoteko *standalone-full.xml* namesto *standalone.xml*. V nadaljevanju bomo uporabljali le nastavitveno datoteko *standalone-full.xml*, ki zažene strežnik Wildfly s celotnim naborom tehnologij platforme Java EE, saj osnovni spletni profil ne vsebuje vseh potrebnih tehnologij.

Strežnik je s tem pripravljen za namestitev EAR aplikacije. Namestitev lahko izvedemo z uporabo spletne konzole na naslovu *localhost:9990*, vendar je ta zamudna v primerjavi z Gradle dodatkom Cargo. Zagon strežnika in namestitev EAR aplikacije tako izvedemo z nalogo *cargoRunLocal*, vendar moramo pred tem nastaviti datoteko *build.gradle* v modulu *strežnik*. Dodati moramo razširitev Cargo ter nastaviti naslednje lastnosti v spremenljivki *cargo*:

- *containerId* - tip aplikacijskega strežnika,
- *port* - vrata strežnika
- *deployable* - arhivi, ki jih namestimo na aplikacijski strežnik
- *local* - pot lokalnega aplikacijskega strežnika in uporabljena nastavitev

Ker uporabljamo lokalen aplikacijski strežnik Wildfly 10.0.0, lastnost *containerId* nastavimo na vrednost *wildfly10x*. Lastnost *local* nastavimo z vrednostjo nastavitvene datoteke in korenskim imenikom lokalnega Wildfly strežnika. Ker želimo namestiti EAR arhiv, v lastnosti *deployable* nastavimo pot do arhiva EAR arhiva v modulu *arhiv poslovne aplikacije EAR*. Gradle nalogama *cargoRunLocal* in *cargoStartLocal* dodamo še odvisnost na nalogo *:app:server:ear:ear*, ki se mora izvesti pred njima, saj ta ustvari arhiv EAR. Strežnik nato lahko poženemo z *cargoRunLocal* ali *cargoStartLocal*. Slednji se bo izvajal v ozadju in ga ustavimo z ukazom *cargoStopLocal*.

6.2.4 Mikrostoritev

Implementirana mikrostoritev za zagon potrebuje že nameščeno podatkovno bazo in ogrodje KumuluzEE, ki smo ga modulu dodali kot odvisnost. Modul *mikrostoritev* vsebuje tri razredi, ki smo jih zasnovali z moduloma *poslovne logike* in *REST komponent*. Da omogočimo izvajanje teh v ogrodju KumuluzEE, moramo dodati še konfiguracijske datoteke za posamezne komponente.

Za delovanje komponente kumuluzee-jpa-hibernate, ki implementira tehnologijo JPA, moramo dodati datoteko *./resources/META-INF/persistance.xml*. Vsebina te je podobna datoteki *persistance.xml* v modulu *shema baze*. Podatkovni vir definiramo v datoteki *./resources/config.yaml*, ki je strukturno enak podatkovnemu viru v aplikacijskem strežniku Wildfly. Da jedro ogrodja KumuluzEE prepozna komponento kumuluzee-cdi-weld, dodamo datoteko *./resources/META-INF/beans.xml*. To komponento potrebujemo, ker v viru izdelkov vstavljamo EntityManager z antacijo @PersistenceContext. Da smo

omogočili uporabo tehnologije JAX-RS, smo morali dodati komponenti kumuluzee-jax-rs-jersey in kumuluzee-servlet-jetty. Slednja za pravilno vzpostavitev potrebuje uvodno spletno stran. To definiramo z datoteko *./resources/webapp/index.html*.

```
...
task buildKumuluzEE {
    dependsOn assemble
    dependsOn copyCompileDependencies
    dependsOn copyCompileClasses
    copyCompileDependencies.mustRunAfter assemble
    copyCompileClasses.mustRunAfter assemble
}

task runKumuluzEE(type: JavaExec) {
    dependsOn assemble
    main = 'com.kumuluz.ee.EeApplication'
    classpath = files(project.buildDir.name + "/kumuluzee/classes") +
        fileTree(dir: project.buildDir.name + "/kumuluzee/dependency")
    environment('PORT', '7080')
}
```

Odsek 6.13: Vsebina datoteke *build.gradle* modula *microservice* z nalogama *buildKumuluzEE* in *runKumuluzEE*

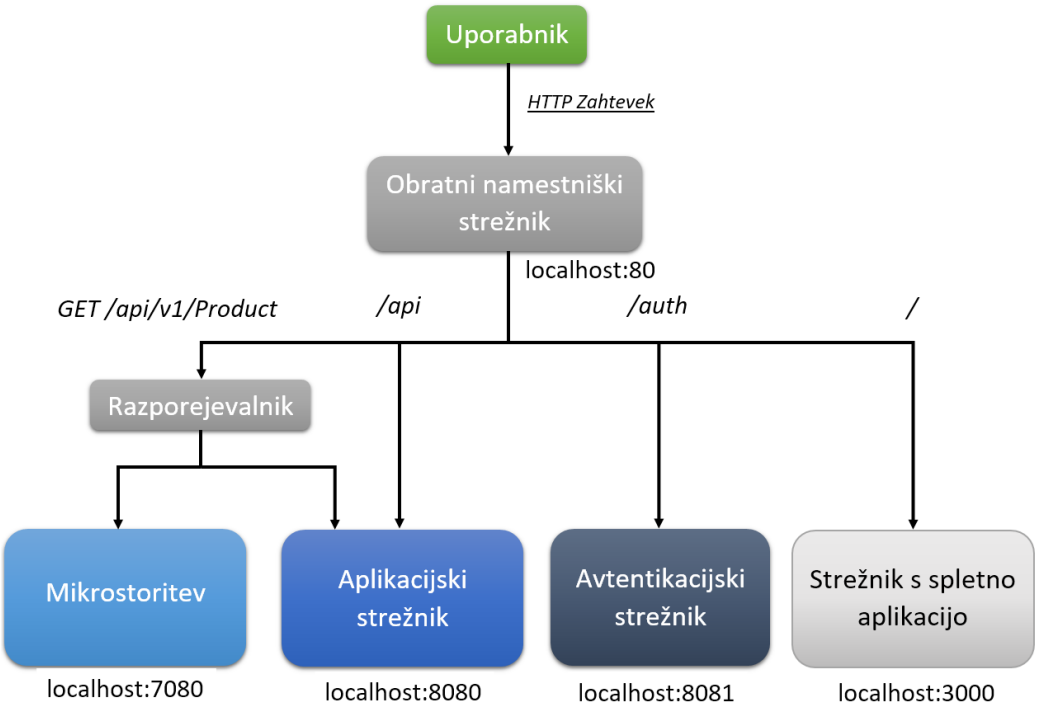
S tem je konfiguracija zaključena. Potrebno je le še zgraditi modul z orodjem Gradle. Privzeta gradnja projekta je za razvijalca slabo pregledna, zato strukturo datotek prilagodimo ogrodju. V imeniku *build* ustvarimo nov imenik imenovan *kumuluzee*, ki vsebuje imenika *classes* in *dependency*. V imeniku *classes* hranimo vso zložno kodo in nastavitve modula *microservice*, medtem ko v imeniku *dependency* hranimo vse odvisnosti. To funkcionalnost implementiramo z nalogo *buildKumuluzEE* prikazano v odseku 6.13. Mikrostoritev nato lahko v mapi *./build/kumuluzee/* poženemo z ukazom *java -cp classes;dependency/* com.kumuluz.ee.EeApplication* ali z nalogo *runKumuluzEE*. Paziti moramo na konflikt uporabljenih vrat (*port*), ki ga lahko spremenimo z okoljsko spremenljivko *PORT*. Privzeto vrednost te spremenljivke smo v nalogi *runKumuluzEE* nastavili na 7080.

6.2.5 Obratni namestniški strežnik

Ker želimo uporabnikom omogočiti dostop do strežnikov preko skupnega vira in se izogniti mehanizmu CORS, uporabimo obratni namestniški strežnik

Nginx. Ta dovoljuje preusmerjanje zahtevkov iz enotnega vira na posamezni strežnik. Hkrati služi tudi kot predpomnilnik odgovorov in razdeljevalnik zahtevkov. Nastavili bomo naslednjo delitev poti in preusmeritev zahtevkov:

- */auth/* - preusmeritev zahtevkov na avtentikacijski strežnik,
- */api* - preusmeritev zahtevkov na aplikacijski strežnik,
- *GET /api/v1/Product* - porazdeljevanje GET zahtevkov namenjenih viru izdelkov na aplikacijski strežnik in mikrostoritev,
- */* - zahtevek preusmeri na statičen strežnik z JavaScript aplikacijo,



Slika 6.4: Prikaz porazdeljevanja zahtevkov na strežniku Nginx

Porazdelitev zahtevkov prikazuje slika 6.4. Privzeto se vsi zahtevki posredujejo strežniku Express z JavaScript aplikacijo. Kadar pot zahtevka vsebuje predpono */api*, strežnik preveri tudi preostale ustrezne daljše poti. V tem

primeru */api/v1/Product* predstavlja daljšo pot z enako začetno predpono. Izbere se nastavitev poti, ki natančneje določa pot zahtevka. Če se pot ujema s predpono */auth* nadaljnji pregled ni potreben, saj ni daljše poti z enako predpono.

Predpomnjenje nastavimo le potema / in */api/v1/Product*, kar najbolj razbremeni informacijski sistem. Predpomnenje enkratnih odgovorov avtentikacijskega strežnika in odgovorov REST vmesnika, ki so različni za posameznega uporabnika, ni smiselno. Posamezen zahtevek strežnik Nginx ohranja za 10 minut, če želimo preskočiti pomnilnik in pridobiti najnovejši odgovor iz strežnika zahtevku dodamo parameter poizvedbe *nocache=true*, vendar se njegova uporaba v produkcijskem okolju navadnim uporabnikom onemogoči. Predpomnjenje lahko zaseda veliko prostora, zato smo ga omejili na velikost 1 GB. Ob zapolnitvi prostora strežnik prične brisati odgovore, ki najdlje niso bili uporabljeni. Odgovori se izbrišejo tudi kadar ti niso uporabljeni za več kot 15 minut. Na podlagi pridobljenih podatkov o prometu lahko kasneje predpomnjenje še bolje prilagodimo posameznemu viru.

Za pravilno posredovanje vpisa v sistem moramo zahtevkom namenjenim avtentikacijskem strežniku Keycloak dodati glave *Host*, *X-Real-IP*, *X-Forwarded-For*, *X-Forwarded-Proto*. Spremeniti je potrebno še nastavitveno datoteko Keycloak strežnika imenovano *standalone.xml*. V njej znački <http-listener> dodamo lastnost *proxy-address-forwarding* z vrednostjo *true*, kar dovoli avtentikacijo preko namestniškega strežnika.

Omenili smo tudi razbremenitev strežnika z uporabo mikrostoritev. Če se nam promet v nekem obdobju poveča, lahko dinamično dodamo ustrezno število mikrostoritev. Ob zagonu mora mikrostoritev svoj obstoj posredovati strežniku Nginx, da jo doda v skupino strežnikov za GET zahtevke izdelkov. Ta funkcionalnost ni vključena v strežnik nginx, zato moramo dodati razširitev ngx_dynamic_upstream [7]. S tem omogočimo spreminjanje razpoložljivih strežnikov z uporabo GET zahtevkov. Za takšno delovanje bi potrebovali še nadzorni program. Ta bi opazoval obremenitev strežnika in mikrostoritev ter na podlagi tega dodajal ali odvzemal mikrostoritve. Ta funk-

cionalnost je ena izmed ključnih lastnosti uporabe mikrostoritev v oblaku. Kljub temu tega ne bomo implementirali, saj za demonstracijo končne rešitve zadostuje uporaba stalne skupine strežnikov.

```
http {
  proxy_cache_path temp/api_temp levels=1:2 keys_zone=api_cache:10m max_size=1g
  inactive=15m use_temp_path=off; # Definicija pomnilnika api_cache
  ...
  upstream getProductLB {
    server 127.0.0.1:8080; # Aplikacijski streznik Wildfly
    server 127.0.0.1:7080; # Mikrostoritev KumuluzEE
  }
  server {
    listen       80;
    server_name  localhost;

    location /api/v1/Product {
      proxy_cache api_cache; # Uporaba pomnilnika
      ...
      limit_except PUT POST DELETE PATCH { # Delitev GET zahtevkov
        proxy_pass  http://getProductLB; # Preusmeritev na skupino getProductLB
      }
      proxy_pass http://127.0.0.1:8080; # Aplikacijski streznik Wildfly
    }
    location /auth/ {
      proxy_set_header Host  $host;
      proxy_set_header X-Real-IP  $remote_addr;
      proxy_set_header X-Forwarded-For  $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto  $scheme;
      proxy_pass http://127.0.0.1:8081; # Avtentikacijski streznik Keycloak
    }
    location /api/ {
      proxy_pass http://127.0.0.1:8080; # Aplikacijski streznik Wildfly
    }
    location / {
      proxy_cache api_cache; # Uporaba pomnilnika
      ...
      proxy_pass http://127.0.0.1:3000; # Streznik Express za slpetni vmesnik
    }
 }
}
```

Odsek 6.14: Vsebina nastavitvene datoteke *nginx.conf* strežnika Nginx

Ena izmed zahtev infrastrukture informacijskega sistema je bila tudi varna povezava med odjemalcem in strežnikom. Predpostavimo lahko, da so povezave znotraj sistema med strežniki in bazo varne, zato moramo zašifrirati le povezavo med odjemalcem in strežnikom Nginx. To storimo z uporabo protokola HTTPS, ki potrebuje certifikat s privatnim in javnim ključem. Certifikat je potrebno kupiti pri certifikatni avtoriteti, ki zagotavlja njegovo veljavnost.

Končno vsebino nastavitvene datoteke *nginx.conf* prikazuje odsek 6.14. V njej na začetku definiramo predpomnilnik imenovan *api_cache*, kar storimo z lastnostjo *proxy_cache_path*. Temu sledi definicija skupine *getProductLB*, ki

vsebuje IP naslov aplikacijskega strežnika in mikrostoritve. Nato definiramo že omenjeno delitev poti z lastnostjo *location*. Poti */api/v1/Product* smo dodali porazdeljevanje GET zahtevkov med strežnike v skupini *getProductLB*, medtem ko preostale zahtevke obdela aplikacijski strežnik. Poti */auth* smo morali dodati tudi že omenjene glave, ki so potrebne za vpis v avtentikacijski strežnik preko obratnega namestniškega strežnika. Ena izmed prednosti uporabe strežnika Nginx je tudi možnost spreminjanja nastavitev tekom izvajanja. Strežnik ob spremembi nastavitev ni potrebno zaustaviti, vendar samo ponovno naložimo nastavitve z ukazom *nginx -s reload*.

Poglavje 7

Vrednotenje razvite rešitve

V tem poglavju bomo ovrednotili naš pristop razvoja informacijskega sistema. Pregledali bomo ustreznost posamezne tehnologije ter ocenili kako ta izpolnjuje zadane naloge. Pri tem bomo izpostavili slabosti ter predlagali možne izboljšave. Osredotočili se bomo na uspešnost razvoja, skaliranje sistema in nadaljnji razvoj.

7.1 Uspešnost razvoja

Naša implementacija informacijskega sistema ustrezno zadovoljuje vsebinske zahteve. Implementirali smo vso potrebno funkcionalnost strank in administratorjev. Ohranjanje prvotnih podatkov naročil smo zagotovili z uporabo razreda BaseEntityVersion, ki spremembe dodaja kot nove vrstice ter pri tem ohranja stare podatke. Ta razred uporabljajo tabele kupcev, izdelkov in naslovov, saj se na njih sklicuje tabela naročil, ki se ne sme spreminjati. Uporabnikom smo z lastnostjo *X-Content* omogočili izbiro vračanja celotne entitete, ko jo ti spreminjajo ali ustvarjajo. Implementirali smo tudi mehanizem lastništva, ki pri vsaki poizvedbi preveri pravice vpisanega uporabnika za izvedbo zahtevka. Prijavo in registracijo uporabnikov smo integrirali z avtentikacijskim strežnikom Keycloak. S tem smo zadostili vsem vsebinskim zahtevam med katerimi smo mnoge tudi nadgradili.

Infrastrukturo informacijskega sistema smo zasnovali z brezplačnimi in odprtokodnimi tehnologijami, ki so zelo fleksibilne, saj lahko brez večjih problemov zamenjamo posamezne tehnologije, če menimo da te niso več ustrezne. Enostavno lahko zamenjamo relacijsko podatkovno bazo, avtentikacijski strežnik, aplikacijski strežnik ter obratni namestniški strežnik. Želena relacijska podatkovna baza mora imeti le svojo implementacijo JDBC gonilnika, medtem ko bi bilo pri menjavi aplikacijskega strežnika potrebno preveriti podprte tehnologije platforme Java EE. Podoben problem bi imeli tudi pri menjavi avtentikacijskega strežnika, saj bi za njegovo integracijo potrebovali ustrezen adapter. Dodajanje novih tehnologij in poslovne logike je enostavno, saj smo strukturo projekta zasnovali modularno in pri tem uporabili fleksibilno orodje Gradle. Za lažje odkrivanje zmogljivosti REST vmesnika smo razvijalcem pripravili tudi OpenAPI dokumentacijo ter Java knjižnico vmesnika. Medtem smo visok nivo varnosti zagotovili z avtentikacijskmi strežnikom Keycloak. Na področju skalabilnosti zahtev nismo presegli, saj bi bilo potrebno implementirati dodatno logiko za dinamično skaliranje. Poslovno logiko na aplikacijskem strežniku in mikrostoritvah bi lahko sicer brez večjih ovir skalirali, vendar bi se kmalu pojavilo ozko grlo pri zmogljivosti podatkovne baze. Da bi rešili ta problem, bi bilo potrebno narediti nekaj večjih sprememb, kljub temu menimo, da je bil razvoj informacijskega sistem uspešen.

7.2 Skaliranje

Ena izmed najpomembnejših lastnosti informacijskih sistemov danes je učinkovitost skaliranja, predvsem zaradi naraščajočega števila uporabnikov. Če sistem vsebuje kritične točke, skaliranje teh ni enostavno, morda celo nemogoče brez večjih arhitekturnih sprememb. Velikokrat so to transakcije, ki si prisvojijo del virov, ki morajo biti sinhronizirani med posameznimi zahtevki. Problem lahko predstavljajo tudi relacijske podatkovne baze, kadar te zahtevajo medsebojno usklajenost. Skaliranje smo razdelili na aplikacijski in podatkovni

sloj, ki ju v nadaljevanju podrobneje preučimo.

7.2.1 Aplikacijski sloj

Tehnologija REST je sama po sebi zelo skalabilna, kljub temu jo lahko implementiramo na več različnih načinov, ki se med seboj razlikujejo po učinkovitosti. V diplomski nalogi smo omenili monolitni način in mikrostoritve. Uporabili smo oba načina, vendar mikrostoritev omogoča le iskanje izdelkov in ne tudi oddaje naročil. S tem načinom bi mnogim uporabnikom zagotovili odzivno iskanje izdelkov s prilagajanjem števila mikrostoritev, če privzamemo, da podatkovna baza ne predstavlja ozkega grla. Temu ni tako pri naročilih, saj strežnik obdela vse zahtevke. Enak problem imamo tudi za vir uporabnikov in naslovov, zato je potrebno poleg mikrostoritev skalirati tudi EAR aplikacijo na aplikacijskem strežniku. V prejšnjih poglavjih smo že omenili slabosti skaliranja EAR aplikacije v primerjavi z mikrostoritvami, zato ob večji potrebi skaliranja preostalih virov priporočamo celoten prestop na mikrostoritve. V kolikor to trenutno še ni mogoče, skaliramo EAR aplikacijo, ki služi zgolj kot začasna rešitev, do popolnega prestopa na mikrostoritve.

V našem primeru smo predpostavili, da je največja obremenitev pri iskanju izdelkov in smo zato implementirali le to mikrostoritev. S tem smo zmanjšali potrebo po skaliranju celotne EAR aplikacije, če predpostavimo, da se strežnik in mikrostoritev izvajata na svoji napravi in strežeta enakemu številu uporabnikov. Ob tej predpostavki in v primeru da je 90% vseh zahtevkov namenjenih iskanju izdelkov, se lahko ob vsakem strežniku izvaja še 9 mikrostoritev. Z uporabo mikrostoritev skaliranje celotne EAR aplikacije tako zmanjšamo za kar 10-krat.

Omeniti je potrebno še avtentikacijski strežnik, ki ga je tudi potrebno skalirati. V našem projektu uporabljamo strežnik Keycloak, ki shranjuje podatke v svojo lokalno bazo. Ta ni ustrezna za skaliranje, saj potrebujemo usklajen podatkovni vir. Problem rešimo z novo neodvisno podatkovno bazo, ki bo hranila podatke vseh uporabnikov. Ko imamo več strežnikov Keycloak

moramo med njimi ustrezno razporediti uporabnike ter spremeniti njihov podatkovni vir, da namesto lokalne baze uporabljajo skupno bazo.

Učinkovitost skaliranja

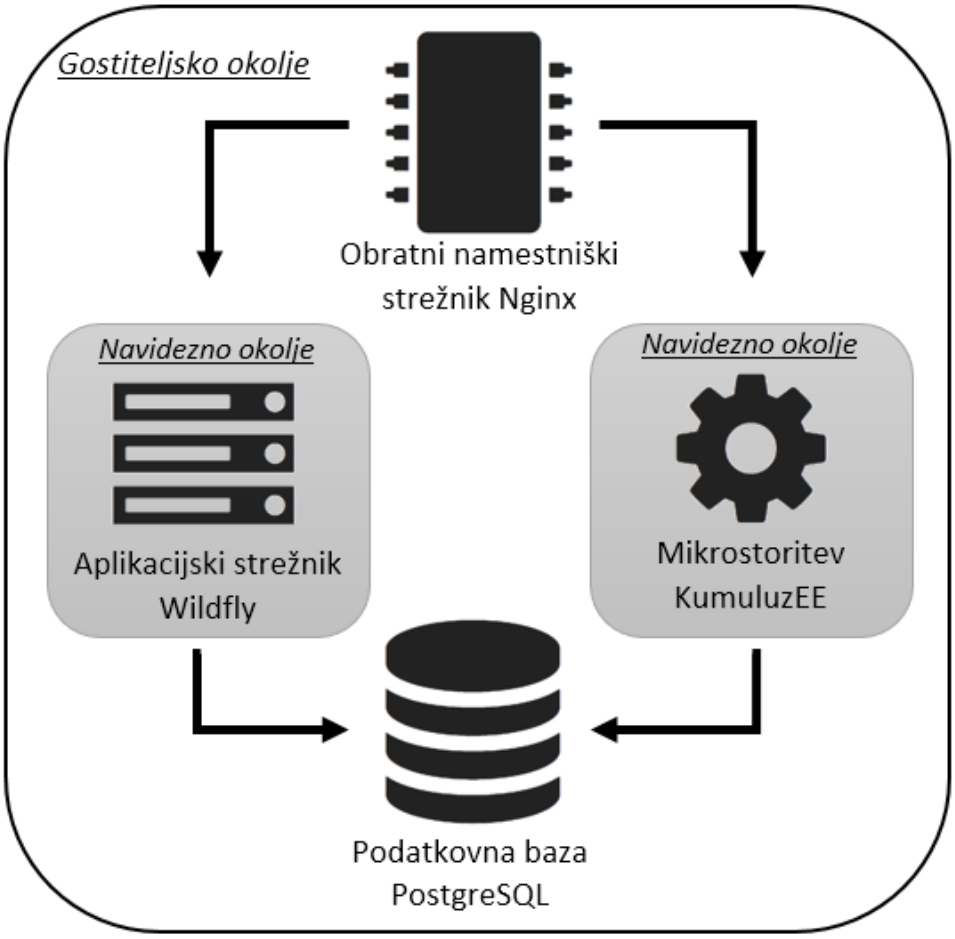
Za podrobnejšo analizo učinkovitosti skaliranja na aplikacijskem sloju informacijskega sistema z dodajanjem strojne opreme smo pripravili zmogljivostne teste. Za njihovo implementacijo smo uporabili orodje Apache JMeter. Ta nam omogoča enostavno simulacijo velikega števila uporabnikov in zahtevkov. Hkrati nudi tudi velik nabor orodij za sestavljanje zapletenih poizvedb. Orodje JMeter med testiranjem izvajamo v načinu brez uporabniškega vmesnika, saj s tem zmanjšamo porabo procesnih virov in povečamo točnost rezultatov. Ker se je simulacija uporabnikov izvajala na isti strojni opremi, so rezultati testiranih aplikacij slabši in ne predstavljajo dejanske zmogljivosti. Pri rezultatih se bomo osredotočili na relativno razliko posamezne in skupne zmogljivosti aplikacijskega strežnika Wildfly in mikrostoritve KumuluzEE.

Testno okolje smo pripravili na računalniku z operacijskim sistemom Windows 10, ki ima sledečo strojno opremo:

- procesor Intel I7-3770K 3.5 GHz - ima vključeni nastavitvi Hyper-threading in Intel Virtualization Technology,
- matična plošča Asus P8Z77-V,
- pomnilnik Corsair Vengeance 1866Mhz 4GB x 4
- disk Samsung SSD 850 EVO 250GB

Za testiranje smo pripravili navidezno okolje s programom VirtualBox, v katerem se bosta izvajala aplikacijski strežnik Wildfly 10.0.0.Final in mikrostoritev KumuluzEE 2.2.0. V navideznem okolju smo namestili namizni (*desktop*) operacijski sistem Ubuntu 16.04 LTS z razvojnim okoljem JDK 1.8.0_144. Posameznemu navideznemu okolju smo tudi omejili računalniške vire na 2 procesorski jedri, 2 GB pomnilnika in 10 GB prostora na disku. S

tem poskušamo simulirati dve enako zmogljivi ločeni fizični napravi ter preprečiti tekmo za pridobitev procesnih virov med aplikacijskim strežnikom in mikrostoritvijo, ko se izvajata hkrati. Test se osredotoča na aplikacijski sloj, zato podatkovno bazo PostgreSQL 9.6 in obratni namestniški strežnik Nginx 1.12.0 namestimo v gostiteljskem okolju. Tako postavitev okolja prikazuje slika 7.1.

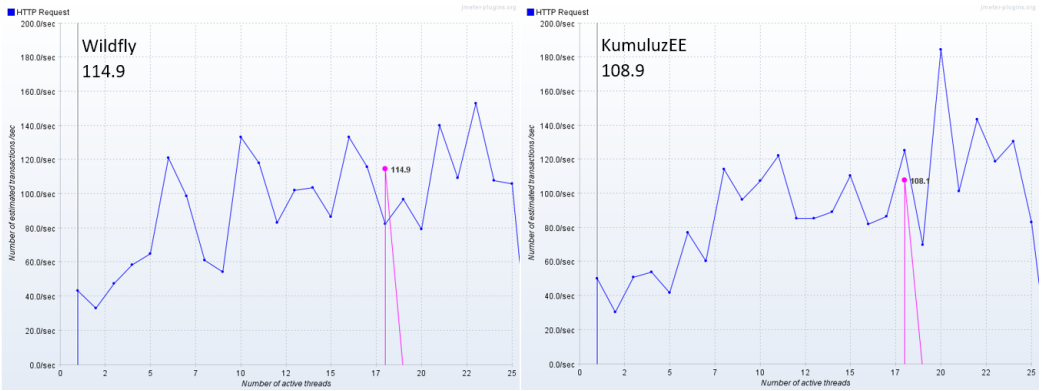


Slika 7.1: Prikaz testnega okolja in komunikacije med obratnim namestniškim strežnikom Nginx, aplikacijskim strežnikom Wildfly, mikrostoritvijo KumuluzEE in podatkovno bazo PostgreSQL

Testirali bomo prepustnost HTTP GET zahtevkov na viru izdelkov. Zahtevek bo tako vseboval pot */api/v1/Product* in parametre poizvedbe:

- *limit* - omejuje število vrnjenih izdelkov in ima tekom testiranja stalno vrednost 10,
- *skip* - predstavlja število preskočenih izdelkov in ima stalno vrednost 0,
- *order* - vsebuje lastnost po kateri urejamo pridobljene izdelke in se med testiranjem naključno spreminja,
- *where* - vsebuje filtre za iskanje izdelkov po vrednostih lastnosti in se med testiranjem naključno spreminja za lastnosti *price* in *discount*,

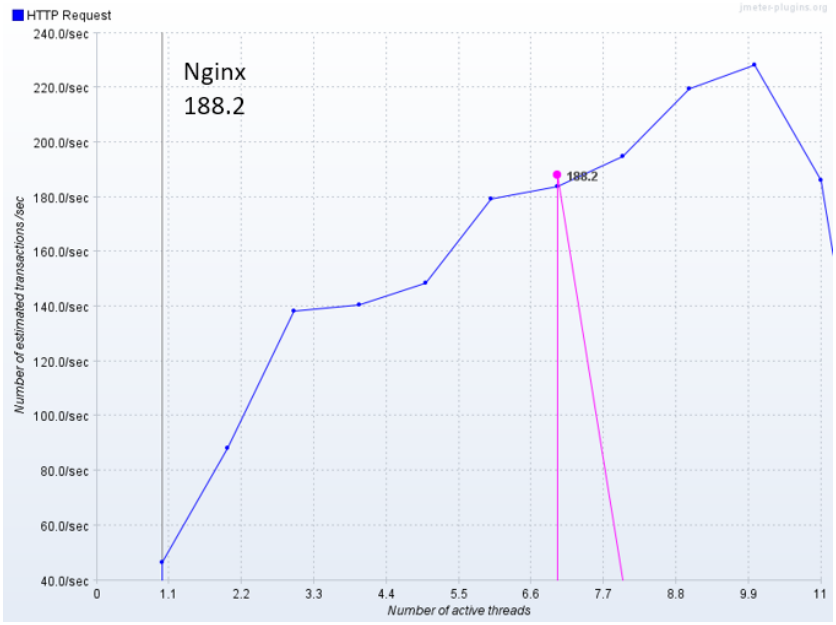
Z orodjem JMeter simuliramo 25 uporabnikov, ki se ustvarijo tekom 5 sekund in zahtevajo obdelavo 30 poizvedb. Sprva testiramo posamezno zmogljivost aplikacijskega strežnika in mikrostoritve. Naša hipoteza je, da bosta implementaciji zelo enakovredni po zmogljivosti, saj obe uporabljata tehnologije platforme Java EE. Grafa na sliki 7.2 prikazujeta obdelano število zahtev-



Slika 7.2: Grafa prikazujeta predvideno število obdelanih zahtevkov na sekundo (*Number of estimated transactions/sec*) v odvisnosti s številom sočasnih uporabnikov (*Number of active threads*) za aplikacijski strežnik Wildfly in mikrostoritev KumuluzEE

kov na sekundo v odvisnosti s številom sočasnih uporabnikov za aplikacijski

strežnik Wildfly in mikrostoritev KumuluzEE. Aplikacijski strežnik Wildfly je skozi test povprečno obdelal 114.9 zahtevkov na sekundo, medtem ko je mikrostoritev KumuluzEE povprečno obdelala 108.1 zahtevkov na sekundo. Relativna razlika je tako približno 6%. Za bolj natančne rezultate bi morali uporabiti skupino stalnih zahtevkov ter te izvajati dlje časa na ločeni strojni opremi. Kljub temu lahko ocenimo, da sta implementaciji po zmogljivosti precej enakovredni in potrdimo hipotezo.



Slika 7.3: Graf prikazuje predvideno število obdelanih zahtevkov na sekundo (*Number of estimated transactions/sec*) v odvisnosti s številom sočasnih uporabnikov (*Number of active threads*) za obratni namestniški strežnik Nginx, ki zahtevke porazdeljuje med aplikacijski strežnik Wildfly in mikrostoritev KumuluzEE

Testiranje izvedemo še z obratnim namestniškim strežnikom Nginx, ki zahtevke enakomerno porazdeljuje med aplikacijskim strežnikom in mikrostoritvijo. Pričakujemo malce slabši rezultat prepustnosti zahtevkov kot skupni rezultat prejšnje meritve (114.9 + 108.1 = 223.0), saj obratni namestniški strežnikom Nginx ne predpomni zahtevkov in porablja del procesnih virov.

Graf na sliki 7.3 prikazuje obdelano število zahtevkov na sekundo v odvisnosti s številom sočasnih uporabnikov za obratni namestniški strežnik Nginx, ko ta porazdeljuje zahtevke med aplikacijski strežnik Wildfly in mikrostoritev KumuluzEE. Skupaj so strežniki med testiranjem povprečno obdelali 188.2 zahtevkov na sekundo, kar predstavlja 41% izboljšavo v primerjavi s posamezno povprečno prepustnostjo (223.0 / 2 = 111.5), vendar je ta za 18% slabša kot njuna skupna prepustnost (223.0). Razlog za to je predvsem uporaba dodatnega strežnika Nginx, ki odvzame del procesnih virov podatkovni bazi PostgreSQL in navideznemu okolju. Za pridobitev bolj relevantnih podatkov bi morali vsak strežnik in mikrostoritev izvajati na svoji strojni opremi ter uporabiti večje število mikrostoritev. To bi lahko prikazali v oblačni platformi, kjer bi zakupili ustrezno število zmogljivih strežnikov in uporabili storitve oblačne platforme za razporejevalnik in podatkovno bazo. Kljub temu menimo, da naši rezultati dobro predstavljajo predvideno učinkovitost skaliranja informacijskega sistema.

7.2.2 Podatkovni sloj

V projektu smo uporabili relacijsko bazo PostgreSQL. Za demonstracijo REST tehnologije podatkovne baze ni bilo potrebno skalirali, temu ni tako v produkcijskem okolju. To še posebej velja za tehnologijo REST, saj se večina operacij izvaja na bazi. Obdelava poizvedb izdelkov na aplikacijskem strežniku ali mikrostoritvi zahteva malo procesnih virov. Potrebno je le pridobiti vrednosti filtrov in sestaviti JPQL poizvedbo. To nato obdela baza, ki predstavlja večinski delež porabe procesnih virov potrebnih za obdelavo zahtevka. Ob skaliranju aplikacije moramo zato dodati tudi ustrezno število podatkovnih baz. Relacijske baze lahko skaliramo na več načinov, pri tem ima vsak svoje prednosti in slabosti.

Za naš projekt bi bilo ob novi mikrostoritvi dovolj podvajanje glavne baze za potrebe branja. Ob vsakih N dodanih mikrostoritvah bi morali dodati novo podatkovno bazo, ki bi vsebovala samo tabelo izdelkov. Pri tem bi morali zagotoviti, da se vsaka sprememba v glavni bazi posreduje tudi

vsem podvojenim bazam. Če se izdelki spreminjajo le redko, bi spremembe lahko posredovali takoj. V nasprotnem primeru, ko imamo opravka z večjim številom sprememb, bi te posredovali periodično s pregledom polj *createdOn* in *editedOn*. Tak način skaliranja povečuje le prepustnost branja podatkov in se v našem projektu lahko uporablja samo za mikrostoritev. Omenili smo tudi skaliranje strežnika, ki potrebuje celotno podatkovno bazo in ne le tabelo izdelkov. V tem primeru bi morali podvojiti celotno bazo in hkrati zagotavljati usklajenost med njimi. Posamezni strežnik entitete lahko dodaja ali spreminja v dodeljeni podatkovni bazi. Tu se pojavi veliko problemov, ki nimajo enostavnih rešitev. Eden izmed problemov je vodenje unikatnega identifikatorja. Naša implementacija uporablja celo število, ki bi ga morali usklajevati med bazami, ko se dodajajo nove entitete. Pri trenutnem načinu dodeljevanja identifikatorja, bi hitro prišlo do kolizije. Temu se izognemo z uporabo UUID identifikatorjev, ki se ustvarijo naključno, vendar je verjetnost kolizije skoraj nič, zaradi njihove velikosti. Identifikator predstavljen s celim številom lahko enolično določa 2^{32} različnih entitet, medtem ko UUID v4 lahko enolično določa kar 2^{122} različnih entitet. Zaradi te velike raznolikosti, verjetnost kolizije lahko zanemarimo. V naši implementaciji bi morali spremeniti le razred BaseEntity in BaseEntityVersion. V razredu BaseEntity bi lastnost *id* predstavili z tipom UUID, enako bi storili v razredu BaseEntityVersion za lastnost *originId*. Če uporabljamo JPA ponudnika Hibernate s podatkovno bazo PostgreSQL, je potrebna sprememba prikazana v odseku 7.1.

```
@Id
@org.hibernate.annotations.Type(type="pg-uuid")
private UUID id;
```

Odsek 7.1: Primer definicije identifikatorja z lastnostjo *id* tipa UUID

Kljub temu prehod iz celega števila na UUID v produkcijskem sistemu ni tako enostaven. Največji problem predstavljajo obstoječe povezave kot je */Product/1*, saj bi te vse postale neveljavne. Vse storitve, ki uporabljajo stari način, bi prenehale delovati, kar si danes podjetja ne morejo privoščiti.

Zato tak prestop naredimo postopoma. Poleg stare verzije bi ponudili novo verzijo REST vmesnika, ki uporablja UUID identifikator. Uporabnikom bi tako morali sporočiti zadnji datum delovanja starega vmesnika, s čimer bi jih spodbudili k uporabi novega. Med tem časom bi morali voditi dve različni podatkovni bazi, kar ni enostavno. Dobro poznavanje potreb informacijskega sistema na začetku razvoja je ključno, da se izognemo podobnim problemom v nadaljevanju.

Omeniti je potrebno tudi povezave do podatkovne baze. Vzpostavitev teh je zamuden postopek, zato JPA tehnologija uporablja bazen povezav (*connection pool*), ki omogoča njihovo ponovno uporabo. Največje število povezav strežnika Wildfly nastavimo v datoteki *standalone.xml*, medtem ko mikrostoritev te podatke hrani v datoteki *config.yaml*. V obeh primerih je največje število povezav nastavljeno na 10. Privzeta vrednost maksimalnega števila povezav relacijske baze PostgreSQL je 100, kar omogoča priklop 10 mikrostoritvam z vsemi možnimi povezavami. Seveda število povezav lahko spremenimo, vendar moramo dobro poznati zmogljivosti strojne opreme, saj s prekomernim povečevanjem povezav lahko tudi poslabšamo odzivnost sistema.

Slika 7.4: Skaliranje celotne in delne podatkovne baze z dvema razdeljevalnikoma povezav

Za naš projekt lahko uporabimo skaliranje baze prikazano na sliki 7.4. Poleg razdeljevalnika zahtevkov dodamo še razdeljevalnika podatkovnih povezav. Te dva skrbita, da so povezave enakomerno porazdeljene med podatkovnimi bazami. To nalogo lahko dodelimo programski rešitvi Pgpool [28]. Ker je podatkovna baza različna med aplikacijskim strežnikom in mikrostoritvijo, potrebujemo dva ločena razdeljevalnika povezav. Eden skrbi za celotne baze, medtem ko drugi skrbi za baze s tabelo izdelkov. Mikrostoritve baz nikoli ne spreminjajo in jih uporabljajo le za branje, zato vse mikrostoritve porazdelimo med baze, ki vsebujejo samo tabelo izdelkov. Aplikacijski strežniki se povezujejo do celotnih baz, ki jih lahko tudi spreminja, zato moramo te med seboj usklajevati. Z uporabo UUID identifikatorja združevanje podatkov ni problematično. Težava je še vedno pri istočasnem spreminjanju iste entitete, kar lahko rešujemo na različne načine. Kadar se ista entiteta spremeni v dveh različnih bazah lahko ohranimo le najnovejšo različico ali najstarejšo še spremenjeno. Če ohranjamo najnovejšo, uporabnik ne bo vedel, da je spremenil drugačno entiteto, kot jo hrani v svojem lokalnem vmesniku. Medtem ko je pri ohranjanju najstarejše spremembe potrebno uporabniku sporočiti, da je bila njegova sprememba zavrnjena. Temu se lahko delo izognemo s čim krajšim časom med usklajevanjem podatkovnih baz.

Relacijske baze se ne skalirajo najbolje, zato kadar potrebujemo visoko skalabilnost, priporočamo uporabo NoSQL podatkovnih baz, kot je MongoDB. Ta namesto definiranih tabel podatke shranjuje v BSON dokument. Relacije hrani z referencami ali z vstavljanjem celotnih entitet (*embedded*). Koncepta združevanja tabel v bazi MongoDB nimamo, zato moramo za vsako referenco v dokumentu ponovno poslati poizvedbo, kar ne velja v primeru, ko je ta že vstavljena. To drastično pospeši delovanje baze, vendar prinese veliko podvajanja podatkov. Poleg tega tudi nima podpore za transakcije in moramo pravilnost podatkov zagotavljati na nivoju aplikacije. Kljub temu, da lastnosti dokumenta niso strogo definirane, nam baza omogoča filtriranje po njihovih vrednostih, saj BSON format omogoča hiter pregled vsebine dokumenta. Tako kot relacijske baze tudi MongoDB omogoča indeksiranje za

hitrejše iskanje.

Prehod med različnimi relacijskimi bazami ne predstavlja nobenih težav s tehnologijo JPA. Temu ni tako pri NoSQL bazah, saj je bila tehnologija JPA zasnovana za uporabo z relacijskimi podatkovnimi bazami. Vseeno so se pojavile rešitve, kot je Hibernate OGM [26], ki uporabljajo tehnologijo JPA z bazo MongoDB. Za njeno uporabo bi bilo sicer potrebno prilagoditi modul *sheme baze*. Ta poseg ni enostaven in bi zahteval veliko sredstev, če bi imeli sistem že v produkciji.

Na trgu je mnogo rešitev, ki poenostavijo skaliranje podatkovnih baz. Ena izmed njih je EnterpriseDB [11], ki uporablja podatkovno bazo PostgreSQL. Hkrati omogoča tudi integracijo z različnimi tipi baz, kot sta dokumentna ali ključ in vrednost. S tem hranjenje podatkov enostavno prilagodimo potrebam informacijskega sistema. Uporaba tega žal ni brezplačna in zahteva letno naročnino.

7.3 Nadaljnji razvoj

Današnji informacijski sistemi se neprestano razvijajo in z novimi tehnologijami poskušajo pridobiti prednost pred konkurenco, kar je še posebej opazno pri spletnih straneh. Implementacija teh mora bit čim bolj enostavna, kar zlahka dosežemo z modularno zasnovo projekta. Z uporabo orodja Gradle strukturo projekta tudi enostavno spreminjamo in jo prilagajamo potrebam uporabljenih tehnologij. Pri tem nismo omejeni na izbiro enotnega programskega jezika, saj Gradle z uporabo razširitev podpira tudi preostale jezike, kot so C, C++, Python, C#, Javascript, itd. To poenostavi implementacijo obsežnih integracijskih testov, ki vključujejo več različnih tehnologij. Avtomatiziramo lahko tudi izdajo arhivov in nastavitev aplikacije v produkcijskem okolju. Gradle tako predstavlja vsestransko orodje pri razvoju projektov. Orodje Gradle je zadostilo vsem potrebam projekta in se enostavno prilagaja novim tehnologijam, zato menimo, da je bila njegova izbira najustreznejša.

V projektu smo poleg monolitne aplikacije uporabili tudi mikrostoritev. S tem smo še dodatno povečali zapletenost projekta, vendar smo jo omejili z generično zasnovo razredov, saj nam ni bilo potrebno implementirati ločene logike. Zasnovani razredi nam omogočajo tudi enostaven prenos preostalih funkcionalnosti, kot sta oddaja naročil ali urejanje računa uporabnika. Naša mikrostoritev le bere podatke iz baze, zato uporaba transakcij ni bila potrebna. Če želimo omogočiti oddajo naročil, moramo dodati transakcije, ki bi jih vključili z uporabo fasadnega zrna, do katerega bi dostopali s CDI anotacijo @Inject. S tako zasnovo projekta zlahka prenašamo obstoječe funkcionalnosti na mikrostoritve ter pri tem dodajmo novo. Posamezne mikrostoritve opravljajo le del funkcionalnosti in lahko med seboj komunicirajo s HTTP protokolom, kar razvijalcem omogoča uporabo različnih tehnologij za njihovo implementacijo. Integracijo teh nam poenostavi orodje Gradle, kljub temu razvoj v drugem programskem jeziku zahteva ponovno implementacijo obstoječe logike. V tem primeru bi morali skrbeti za dve različni izvorni kodi ter ju usklajevati, zato odsvetujemo uporabo dodatnega jezika za implementacijo že obstoječe funkcionalnosti. Tudi pri implementaciji nove funkcionalnosti moramo imeti tehten razlog za uporabo dodatnega jezika.

Poglavje 8

Sklep

V diplomski nalogi smo uspešno razvili informacijski sistem enostavne spletne trgovine. Zadostili smo vsem postavljenim vsebinskim zahtevam, ki smo jih implementirali z bogatim REST vmesnikom. Uporabnikom smo omogočili napredno iskanje izdelkov, ki jih tekom nakupa lahko hranijo v košarici. Te lahko poljubno urejajo vse do oddaje naročila, za kar potrebujejo aktiven uporabniški račun, ki ga ustvarijo na avtentikacijskem strežniku. Informacijski sistem potrebuje za delovanje tudi uporabnike v vlogi administratorja, ki lahko ureja prodajne izdelke in pregleduje oddana naročila.

Pri razvoju smo si pomagali z orodjem Gradle, katerega funkcionalnost smo prilagodili posameznim modulom. Podatkovno shemo baze smo razvili s tehnologijo JPA v modulu *shema baze*, ki je združljiva z obsežnim naborom relacijskih podatkovnih baz, med katerimi smo izbrali PostgreSQL. Poslovno logiko smo implementirali v modulu *poslovna logika*, ki smo jo postavili za fasadna poslovna zrna s tehnologijo EJB v modulu *poslovna zrna*. REST vmesnik smo implementirali z generičnimi razredi modula *REST komponente*, ki temelji na tehnologiji JAX-RS. Modula *poslovna zrna* in *REST vmesnik* smo nato z vsemi potrebnimi knjižnicami zapakirali v arhiv EAR, ki predstavlja glavno funkcionalnost sistema. Arhiv smo namestili na aplikacijski strežnik Wildfly, ki vsebuje vse potrebne tehnologije za izvajanje aplikacije. Strežnik smo morali integrirati še z avtentikacijskim strežnikom

Keycloak, preko katerega se uporabniki vpisujejo v sistem. Za bolj učinkovito skaliranje smo razvili tudi mikrostoritev v ogrodju KumuluzEE, ki podpira tehnologije platforme Java EE, zato smo pri razvoju ponovno uporabili že implementirano funkcionalnost. Razviti informacijski sistem tako sestavljajo podatkovna baza PostgreSQL, avtentikacijski strežnik Keycloak, aplikacijski strežnik Wildfly, mikrostoritev KumuluzEE ter obratni namestniški strežnik Nginx, ki izpostavlja vse storitve preko enotnega vira. Za demonstracijo uporabe smo dodali še spletni vmesnik razvit s tehnologijo JavaScript v ogrodju React, ki ga streže strežnik Express.

Analizirali smo uporabljene tehnologije in njihovo ustreznost pri reševanju zadanih problemov. Podrobneje smo predstavili problem skaliranja tako na aplikacijskem kot na podatkovnem sloju. Slednji se skalira veliko težje in ga je potrebno skrbno načrtovati že v začetku razvoja, medtem ko smo z zmogljivostnimi testi na aplikacijskem sloju dokazali učinkovitost skaliranja informacijskega sistema z dodajanjem strojne opreme. Za bolj realistične rezultate bi morali sicer povečati obsežnost testov in vključiti večje število mikrostoritev, vendar kljub temu menimo, da rezultati ne bi bil veliko slabši.

Razvoj informacijskega sistema obsega širok spekter znanj in tehnologij, zato se ta lahko hitro zaplete in predstavlja težavo tudi izkušenim razvijalcem. V diplomski nalogi smo predstavili razvoj enostavnega informacijskega sistema, vendar smo vseeno naleteli na kopico problemov, ki smo jih brez večjih težav odpravili. Težavnost se drastično poveča pri razvoju večjih informacijskih sistemov, ki morajo nuditi obsežno funkcionalnost ter hkrati biti odzivni in skalabilni. Kljub temu bi bila naša modularna delitev projekta še vedno ustrezna, saj bi jo lahko enostavno prilagodili. Enako velja tudi za uporabljene tehnologije, saj jih lahko zamenjamo, če te niso več ustrezne. Na podlagi pridobljenih izkušenj menimo, da so uporabljene tehnologije in prakse za razvoj naprednejših informacijskih rešitev primerne tako za začetnike, kakor tudi za že izkušene razvijalce. Celotna izvorna koda projekta je javno dostopna na strani `https://github.com/jgasperlin/is`.

Literatura

[1] agamemnus. react-boilerplate. Dosegljivo: https://github.com/react-boilerplate/react-boilerplate. [Dostopano: 13.7.2017].

[2] Apache. Ant. Dosegljivo: http://ant.apache.org/. [Dostopano: 10.4.2017].

[3] Apache. Maven. Dosegljivo: https://maven.apache.org/. [Dostopano: 10.4.2017].

[4] M. Belshe, R. Peon, and M. Thomson. Hypertext transfer protocol version 2 (http/2). Dosegljivo: http://www.rfc-editor.org/rfc/rfc7540.txt. [Dostopano: 1.9.2017].

[5] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext transfer protocol – http/1.0. Dosegljivo: http://www.rfc-editor.org/rfc/rfc1945.txt. [Dostopano: 10.8.2017].

[6] callemall. material-ui. Dosegljivo: https://github.com/callemall/material-ui. [Dostopano: 1.9.2017].

[7] cubicdaiya. Dynamic upstream for nginx. Dosegljivo: https://github.com/cubicdaiya/ngx_dynamic_upstream. [Dostopano: 13.7.2017].

[8] DB-Engines. Db-engines ranking. Dosegljivo: https://db-engines.com/en/ranking. [Dostopano: 13.7.2017].

[9] L. Dusseault. Patch method for http. Dosegljivo: https://github.com/OAI/OpenAPI-Specification. [Dostopano: 21.4.2017].

[10] L. Dusseault and J. Snell. Patch method for http. Dosegljivo: http://www.rfc-editor.org/rfc/rfc5789.txt, note = [Dostopano: 10.8.2017].

[11] EnterpriseDB. Enterprisedb homepage. Dosegljivo: https://www.enterprisedb.com/. [Dostopano: 1.8.2017].

[12] Facebook. React homepage. Dosegljivo: https://facebook.github.io/react/. [Dostopano: 3.7.2017].

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Dosegljivo: http://www.rfc-editor.org/rfc/rfc2068.txt. [Dostopano: 10.8.2017].

[14] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. Dosegljivo: http://www.rfc-editor.org/rfc/rfc7231.txt. [Dostopano: 10.8.2017].

[15] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. Dosegljivo: http://www.rfc-editor.org/rfc/rfc2616.txt. [Dostopano: 10.8.2017].

[16] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. Dosegljivo: http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf. [Dostopano: 12.4.2017].

[17] Gradle. Gradle. Dosegljivo: https://gradle.org/. [Dostopano: 10.4.2017].

[18] Arun Gupta. *Java EE 7 Essentials Enterprise Developer Handbook*. O'Reilly Media, 2013.

[19] Richard Heeks. Failure, success and improvisation of information systems projects in developing countries. Dosegljivo: `http://unpan1.un.org/intradoc/groups/public/documents/NISPAcee/UNPAN015601.pdf`. [Dostopano: 10.8.2017].

[20] JBoss. Keycloak homepage. Dosegljivo: `http://www.keycloak.org/`. [Dostopano: 3.7.2017].

[21] JBoss. Wildfly homepage. Dosegljivo: `http://wildfly.org/`. [Dostopano: 3.7.2017].

[22] Frank Jennings, Matjaz B. Juric, Poornachandra Sarang, and Ramesh Loganathan. *SOA Approach to Integration*. Packt Publishing, 2013.

[23] Kumuluzee. Kumuluzee homepage. Dosegljivo: `https://ee.kumuluz.com/`. [Dostopano: 3.7.2017].

[24] Sam Newman. *Building Microservices Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[25] Nginx. Nginx homepage. Dosegljivo: `https://www.nginx.com/`. [Dostopano: 3.7.2017].

[26] Hibernate OGM. Hibernate ogm homepage. Dosegljivo: `http://hibernate.org/ogm/`. [Dostopano: 1.8.2017].

[27] Oracle. Design goals of the java programming language. Dosegljivo: `http://www.oracle.com/technetwork/java/intro-141325.html`. [Dostopano: 25.4.2017].

[28] PGPool. Pgpool homepage. Dosegljivo: `http://www.pgpool.net/`. [Dostopano: 1.8.2017].

[29] Dimitri Ponomareff. Introducing agile scrum xp and kanban. Dosegljivo: `https://www.slideshare.net/dimka5/introducing-agile-scrum-xp-and-kanban`. [Dostopano: 25.8.2017].

[30] postgreSQL. Postgresql homepage. Dosegljivo: https://www.postgresql.org/. [Dostopano: 3.7.2017].

[31] PYPL. Popularity of programming language. Dosegljivo: http://pypl.github.io/PYPL.html. [Dostopano: 25.4.2017].

[32] Roger Smith. Why java is the most popular programming language. Dosegljivo: http://www.theserverside.com/feature/Why-Java-is-the-most-popular-programming-language. [Dostopano: 25.8.2017].

[33] Stackify. What are crud operations. Dosegljivo: https://stackify.com/what-are-crud-operations/. [Dostopano: 25.8.2017].

[34] w3schools. Json vs xml. Dosegljivo: https://www.w3schools.com/js/js_json_xml.asp. [Dostopano: 25.8.2017].